15 | The Memory Hierarchy (Chapter 8)

We have levels of memory used to improve performance:

Cache memory	Fast, expensive (used for speed)
Main memory	(AKA RAM)
Virtual memory	Slow, less expensive (used for greater amount of memory, which often results in speeding things up)

Types of memory (*note:* PROM and ROM are often used in devices that need to be booted up by software. The devices come with a PROM and ROM chip to boot it):

Random access memory (RAM)	you have direct access to any byte in that memory; no need to search, you just specify the address. you can change and write to RAM. RAM is volatile, thus it requires power
Read only memory (ROM)	you can read from this memory, but you can't change; once it's been created, you can't change. this is also random access
Programmable ROM (PROM)	you can store a program in it; once you've stored the program, you cannot be changed.
Erasable Programmable ROM (EPROM)	in case you need to change PROM and put a new program into it; for example, if a new version of a device comes out, and you need to change the software driver for that device, and here would come with an erasable PROM. still, it's a ROM in the sense that while the device is running, it cannot write into that memory.
Flash memory	solid-state memory, typically slower than RAM, but NOT volatile (flash memory doesn't require power)

A storage device is **volatile** if the stored data is lost when power is shut off. Permanent storage devices such as disks and flash memory aren't volatile, but RAM and cache memories are.

Memory access time is slow, compared to clock cycle time. About 1000 add instructions can be executed in the time required for a lw or sw instruction (since both instructions need to access RAM, and asking RAM for a word or a byte of memory, it will take a long time to come up with that information).

Memory latency time required for the memory to respond to a read/write request

Much of what we're doing in the memory hierarchy is designed to reduce memory latency. What are some strategies we can use?

- **Cache memory**
  - Fast, but expensive
  - Smaller than RAM
  - Some parts of RAM are copied to cache, as needed
    - If a program needs to access a particular byte of RAM, if there's a copy of that in the cache, it only needs to go to the cache (considerably speeds things up)

- **Virtual memory**
  - Expansion of RAM (moreso expand than speeding RAM up)
    - If we don't have enough memory in our computer, we'll give it VM
  - Uses a peripheral device, such as disk or flash memory
  - However, this slows RAM
    - If you try to access a word of virtual memory that is not in RAM, it will have to go to that slow disk or flash memory to copy that word into RAM and then we have access to it

Memory/storage technologies, fast/expensive at the top and slowest/cheapest at the bottom:

Technology	**Usage**	**Volatile**
FPGA (Field Programmable Gate Array)	CPU Registers (fastest storage devices)	Y
SRAM (Static RAM) - *writable*	Cache	Y
DRAM (Dynamic RAM)	Main memory	Y
Flash (removable, ex. flash drive, or not)	Secondary storage	N
Solid state disk (flash)	Secondary storage or virtual memory	N
Fixed magnetic disk (rotating disk)	Secondary storage or virtual memory	N

	(*cannot be used to store permanent files such as Word documents or something since a portion is reserved for virtual memory*)	
Optical disk	Secondary storage	N
Removable magnetic disk (*not seen as much anymore*)	Secondary storage	N

Cache Memory

- Slower access time than CPU registers
- Faster access time than RAM
- Maintains a copy of RAM data in fixed size chunks called blocks
- Block size could be 256 bytes
- Typical cache size could be 4K = 2^4 blocks = 16 blocks
- Each cache block is an exact copy of a block in RAM

Program execution: *lw* instruction (copying a word from RAM into a register)

Is the word to be loaded already in cache?

- Yes - **cache hit**: no need to access RAM
- No - **cache miss**: its block must be copied from RAM to cache

Program execution: *sw* instruction (writing from CPU register into RAM)

Cache block must eventually be copied back to RAM (to make sure RAM & cache agree)

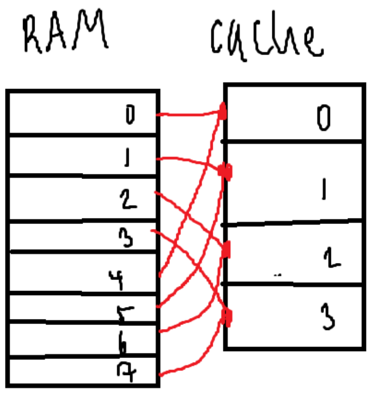
- On every write to cache, block is copied back to RAM to ensure agreement
- Use a **'dirty' bit** which is set when the cache block is changed
    - When there's a write to a cache block, the dirty bit is set to 1
    - When that block is to be replaced by another, if the dirty bit is 1, the block is copied back to RAM, but if it's 0, no need to copy it back

There are two ways to implement cache:

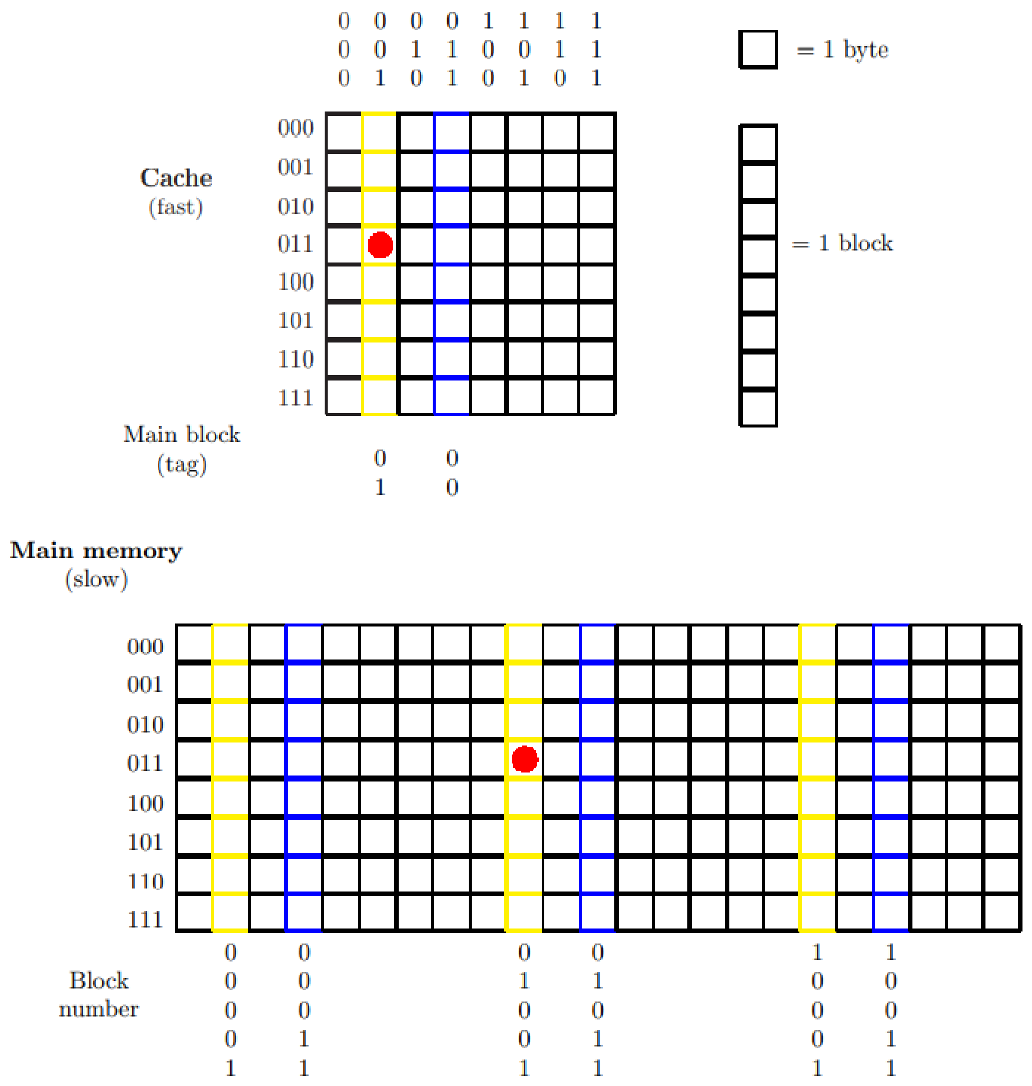
Direct-mapped cache: Each block of RAM is mapped to a specific cache block

- Since RAM bigger than cache, there will be several RAM blocks that map to a given cache block (shown right)

- **Example:** As seen, there are 8 RAM blocks and 4 cache blocks; mathematically, we would say block *n* of RAM maps to block *n* % (mod) 4 of cache
    - The mod is used to indicate remainder of division, so *n* / 4 would give you a division (integer answer), and % would give you that remainder



- **Example:** Direct-mapped cache memory. 256 byte RAM, 8 bytes per block, 8 blocks of cache, 32 RAM blocks.



    - (Just saying, we always show our bytes here vertically; low-order top)

- Here we're showing only about 15 blocks of ram
- The diagram shows which blocks in RAM correspond to the block in cache

- The tags correspond to the high order two bits of RAM, so if we need to write the block back to RAM, we need to know which yellow block to write it to. The tag tells us that. **The format** is tag, cache block, and byte (note the size of a block).

- The **low-order 3 bits of a RAM** (second picture) address (the numbers shown on *y*-axis) show the byte within a block, bytes 0...7 within each block. Then, the **high order 5 bits** (shown in the *x*-axis) are the block number, so block 0...31

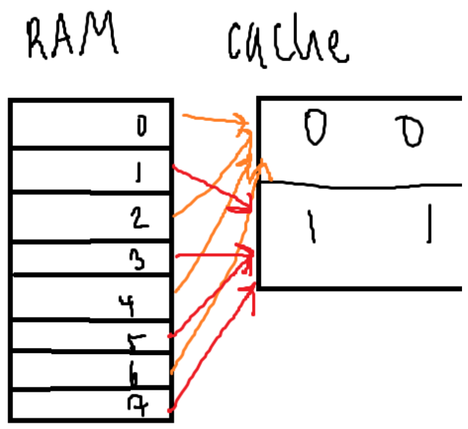
- The **low-order 3 bits of cache** (first picture) address (numbers shown on *y*-axis) indicate the byte within a block, 0...7, and **the high order 3 bits** (shown in the top *x*-axis) indicate the block number within cache, 0 through 7

- So, for the red dot (corresponds to each other), **load from 0100 1011** in RAM to **01 001 011** in cache. Note how the addresses are the same, just spaced differently. The one in the cache is a copy of the one in RAM.

- How about: **store into 1000 1011** in RAM to **10 001 011** in cache. The low order 3 bits of the address are 011, so it's going to be in the same row as the red dot. The high order 5 bits are 10 001, so we're talking about the 3rd yellow column. However, problem: this byte is not in cache! Solution: the whole block would have to be copied into cache, and then we would have to write into that byte of cache, changing that byte of RAM.

Associative cache: Each block of RAM is mapped to a *set* of cache blocks

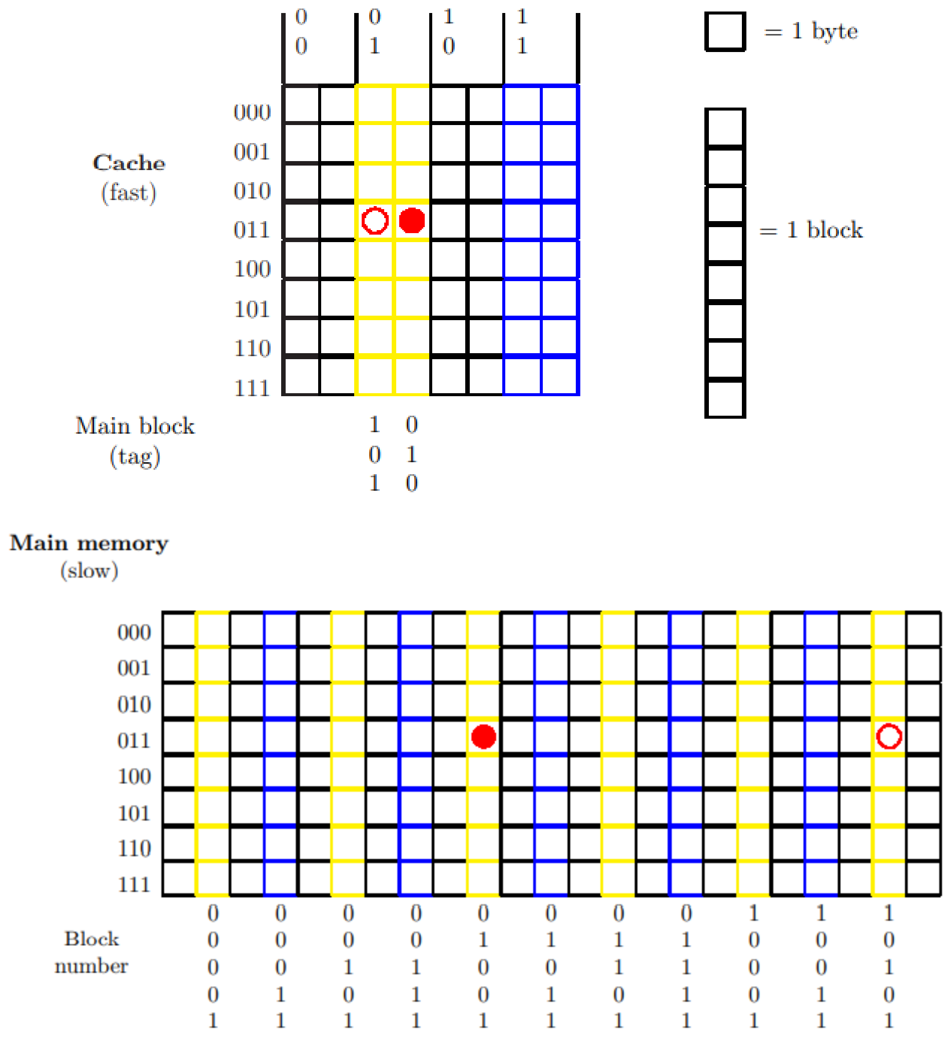
- **Example:** RAM blocks 0, 2, 4, and 6 map to cache set 0. RAM blocks 1, 3, 5, and 7 map to cache set 1.

  - The RAM block can be copied into any of the cache blocks in the set; if we access RAM block 0, it could be copied into the first cache block in set 0 or the second (both are 0's, as you can see)

  - *n* blocks in a set = ***n*-way associative cache**
  - In this example, RAM block *b* matches to Cache Set *b* % *n*

- **Example:** 2-way associative cache memory. 256 byte RAM, 8 bytes per block, 8 blocks of cache, 32 RAM blocks, 2 blocks per set.



  - **RAM address example:** 101 01 011 (tag, cache set, byte)

  - The red dot, let's **load from 1010 1011** = **101 <u>01</u> 011** (regroup it). Which block of ram is it in? It's in the 10101 group. We're talking about the hollow red dot. It's going to get loaded in *one of the two* bytes of cache. Let's assume it gets loaded into the one specified in the cache diagram. So that is an exact copy of the one in RAM.

  - Then we have another **load from 0100 1011** = **010 <u>01</u> 011**. This is going to map to the same cache block (filled red dot), into the same set (notice the underlined portion and the format of the first bullet). So, there will NOT be a cache miss, because we have a set of 2 slots here for the byte being loaded. If there were to be a 3rd reference to the same set, then one of them would have to be replaced (and that would be a cache miss).

- Suppose now there is a reference to **load from** **0000 1111** = **000 01 111**. This is a cache miss for reasons described above. Set is full, so one of the blocks must be copied back to RAM. Which one should be copied and replaced? There are three ways of choosing this:

    - **LRU** - Least Recently Used
    - **FIFO** - First In, First Out
    - **Random** selection (choose two blocks at random and copy back to RAM)

- **Thrashing** is when almost every memory reference results in a cache miss

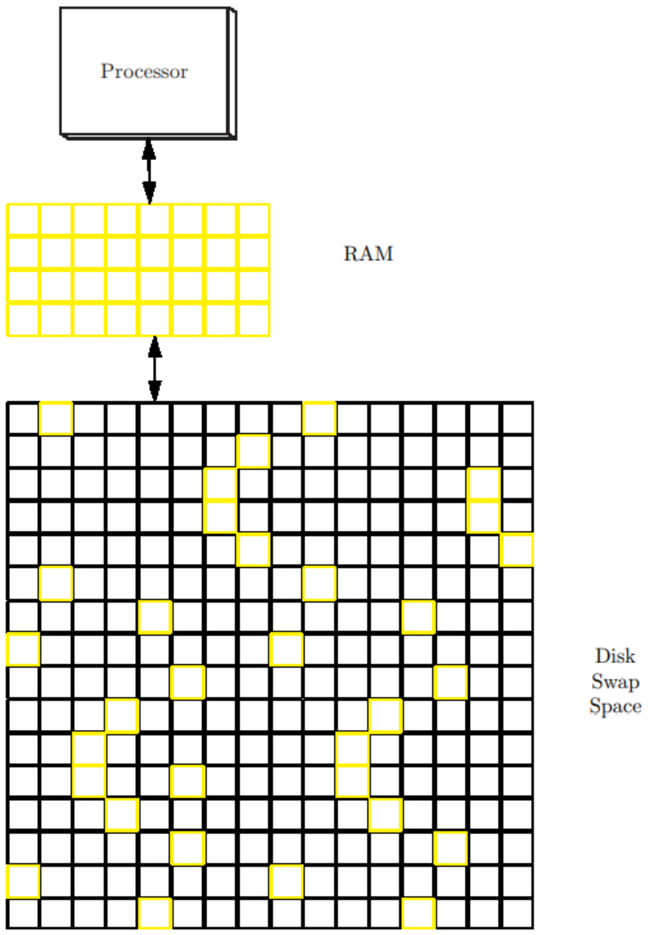
    - If every memory reference is a cache miss, then blocks will be copied back and forth to RAM on every memory reference, and the cache memory won't provide an advantage here
    - Random strategy is best to prevent this
    - No algorithms will be optimal

Virtual Memory

- Allows for expansion of RAM to exceed the physical capacity of RAM
- Uses a magnetic disk, flash memory, or other peripheral storage to increase available memory to a running program
- Similar in concept to cache memory, but doesn't speed up program execution, but does help applications which are 'memory bound' (need a certain amount of memory)

RAM is organized into pages. In our example, one page is 8K bytes. Part of the disk or flash memory is reserved for **swap space**, which represents the entire address space for programs.

The following is a diagram of a virtual memory system for an **example**:

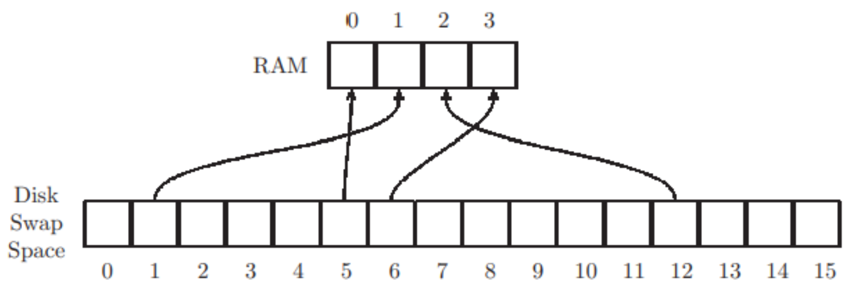


- In RAM we have 32 pages = 2^5 pages
  - Each square represents 1 page (8K = 2^{13} bytes)
  - Multiplying these numbers gives us 256K bytes ($2^{13} * 2^5 = 2^{18}$)
  - Also RAM is one dimensional, really

- In virtual memory, we have 256 pages
  - Really is linear (even though it's shown as 16 by 16)
  - There are 2M bytes in virtual memory ($2^{13} * 2^8 = 2^{21}$)
  - Virtual memory address is 21 bits (note this is the same as the exponents)

Let's try a smaller **example** (see right):

- 4 pages in RAM, 16 pages in virtual memory



- We have references to pages 5, 1, 12, and 6 in that order (so 5 is copied into page 0, telling us that things are copied into RAM in that order from the first available page)

- In this example, page size is 8K = 2^{13} bytes
- **RAM address is 15 bits** since we have 4 pages (thus $2^2 * 2^{13} = 2^{15}$)
- **Virtual memory size is 128K bytes**, since 16 (from 16 pages) * 8K = $2^4 * 2^{13} = 2^{17}$)
- **Virtual memory address is 17 bits** since that exponent

- Thus, we can organize the 17 bits address like this:

1100	0 0001 0101 1000
page number (4)	**page offset (13)**

  This example would be page 5 in the disk swap space, while the offset refers to the specific byte within that page. The address for a system with 16 pages in virtual memory, and 8K bytes in each page, is 0x18158.

- If we put another reference (for example, to page 14), one of the RAM pages needs to be replaced by page 14. If that page had been written, it needs to be copied back to virtual memory. To do that, we need to know where that page came from. So, we'll use a page table (see right).

Disk	RAM
5 = 0101	0
1 = 0001	1
12 = 1100	2
6 = 0110	3

- How is a page selected for replacement for references to pages 5, 1, 12, 6, 5, 14? Let's use the **page replacement algorithm**

  - LRU (Least Recently Used) → replace page 1
  - FIFO (First in, First Out) → replace page 5
  - Random → choose any page at random (works out well)

  - There is no optimal algorithm (which would choose the page which won't be needed for the longest time)
  - If there are a lot of page faults, that's **thrashing**

- A **page fault** is the term used when the system must copy a page from the disk swap space into RAM (when one of the pages, like 5, 1, 12, or 6 needs to be replaced).

- Note that Cache Memory < RAM < Virtual Memory (bigger than, less than), which is shown in the demos

Locality

- Applies at both the cache and virtual memory level
    - Cache memory is capable of improving run-time efficiency
    - Virtual memory is capable of expanding the memory's capacity without a significant degradation of performance
    - Data Memory and Instruction Memory are separate

    - **Memory unit** means either cache block or virtual memory page
    - **Memory fault** means either cache miss or page fault

    - **Data locality** has to do with memory references for data (like lw and sw)
        - When successive lw and sw refers to locations in different places, we can expect memory faults w/ poor data locality

    - **Instruction locality** is affected by transfer of control (branch/jump instructions)
        - When a program executes sequentially, good locality since successive references to instruction memory will generally be the same memory units
        - Memory faults happen if instructions are various and widely separated when fetching an instruction to the Instruction Register
        - Not common since most loops tend to be fairly local
        - ***switch*** or ***branch table*** in a loop could cause poor locality

    - **Temporal locality** is good when memory units accessed are done so subsequently soon thereafter (can be good locality even if it has poor **spatial locality**, which is the degree of which memory references are located in proximity with respect to the number of cache blocks/virtual memory pages, AKA is it using all the space?)

- To what extent are memory references localized?
    - Is a program referencing memory locations that are near each other or those that are dispersed throughout memory?

    - **High degree of locality = few cache misses, few page faults (more efficient)**
        - References are mostly in the same page with good locality, meaning there doesn't have to be many chances of replacing pages in RAM

        - This program will run fast, since memory location references are near each other; choose an algorithm with good locality

    - **Poor locality = many cache misses, many page faults (thrashing will occur)**
        - The references are more scattered, meaning more chances for page faults
        - Pages would keep getting swapped in and out of virtual memory to replace the pages already in RAM, slowing down the program

    - We can change our programs to have improved locality