CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 29: Caches Part 1

Instructors: Dan Garcia, Justin Yokota

Computing in the News

Artificial Intelligence: UNESCO calls on all Governments to implement Global Ethical Framework without delay | UNESCO

"UNESCO is concerned by many of the ethical issues raised by these innovations, in particular discrimination and stereotyping, including the issue of gender inequality, but also the fight against disinformation, the right to privacy, the protection of personal data, and human and environmental rights.

Industry self-regulation is clearly not sufficient to avoid these ethical harms, which is why the Recommendation provides the tools to ensure that AI developments abide by the rule of law, avoiding harm, and ensuring that when harm is done, accountability and redressal mechanisms are at hand for those affected."

Agenda

- Review
  - Binary Prefixes
  - Memory Hierarchy
- Intro to Caches
- Fully Associative (FA) Caches

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

- Common-use prefixes (all SI, except K != k in SI)
- Confusing! Common usage of “kilobyte” means 1024 bytes, but the “correct” SI value is 1000 bytes
- Hard Disk manufacturers & Telecommunications are the only computing groups that use SI factors
  - What is advertised as a 1 TB drive actually holds about 90% of what you expect
  - A 1 Mbit/s connection transfers 106 bps.

Chart

Name	Abbr	Factor	SI size
Kilo	K	2^{10} = 1,024	10^{3} = 1,000
Mega	M	2^{20} = 1,048,576	10^{6} = 1,000,000
Giga	G	2^{30} = 1,073,741,824	10^{9} = 1,000,000,000
Tera	T	2^{40} = 1,099,511,627,776	10^{12} = 1,000,000,000,000
Peta	P	2^{50} = 1,125,899,906,842,624	10^{15} = 1,000,000,000,000,000
Exa	E	2^{60} = 1,152,921,504,606,846,976	10^{18} = 1,000,000,000,000,000,000
Zetta	Z	2^{70} = 1,180,591,620,717,411,303,424	10^{21} = 1,000,000,000,000,000,000,000
Yotta	Y	2^{80} = 1,208,925,819,614,629,174,706,176	10^{24} = 1,000,000,000,000,000,000,000,000

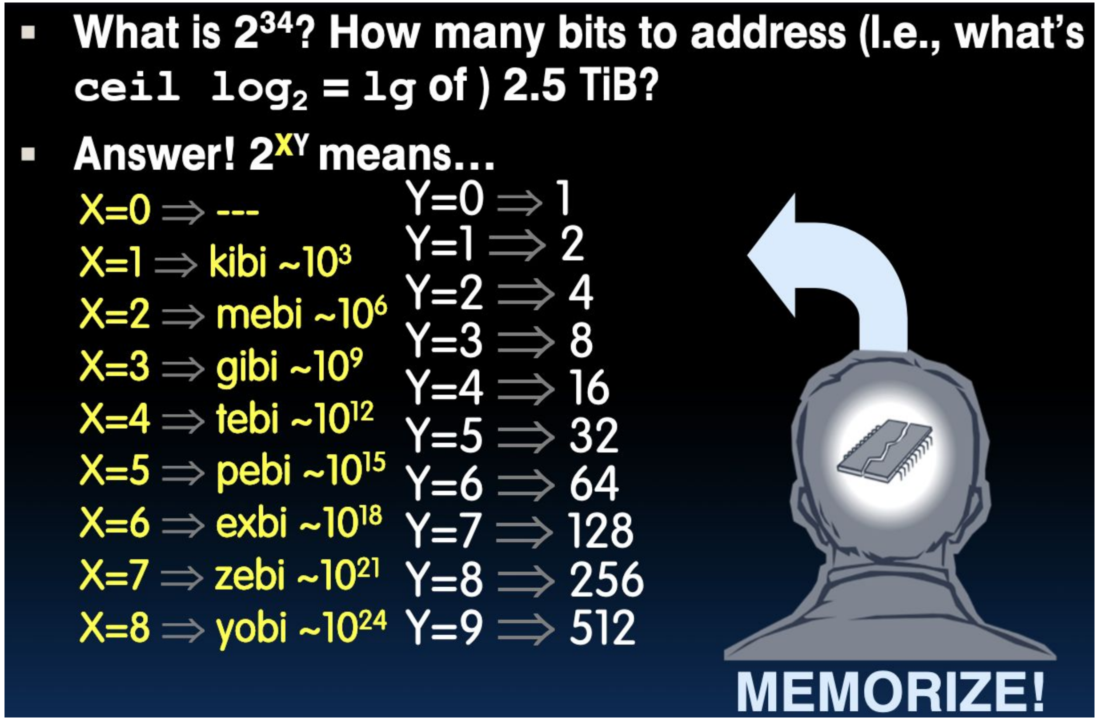
kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

- IEC Standard Prefixes

Name	Abbr	Factor			
kibi	Ki	2^{10} = 1,024	pebi	Pi	2^{50} = 1,125,899,906,842,624
mebi	Mi	2^{20} = 1,048,576	exbi	Ei	2^{60} = 1,152,921,504,606,846,976
gibi	Gi	2^{30} = 1,073,741,824	zebi	Zi	2^{70} = 1,180,591,620,717,411,303,424
tebi	Ti	2^{40} = 1,099,511,627,776	yobi	Yi	2^{80} = 1,208,925,819,614,629,174,706,176

- International Electrotechnical Commission (IEC) in 1999 introduced these to specify binary quantities.
- Names come from shortened versions of the original SI prefixes (same pronunciation) and bi is short for “binary”, but pronounced “bee” :-(
- Now SI prefixes only have their base-10 meaning and never have a base-2 meaning.

The way to remember #s



- What is 2^{34}? How many bits to address (I.e., what's `ceil` $\texttt{log}_2$ = `lg` of) 2.5 TiB?
- Answer! 2^{XY} means…

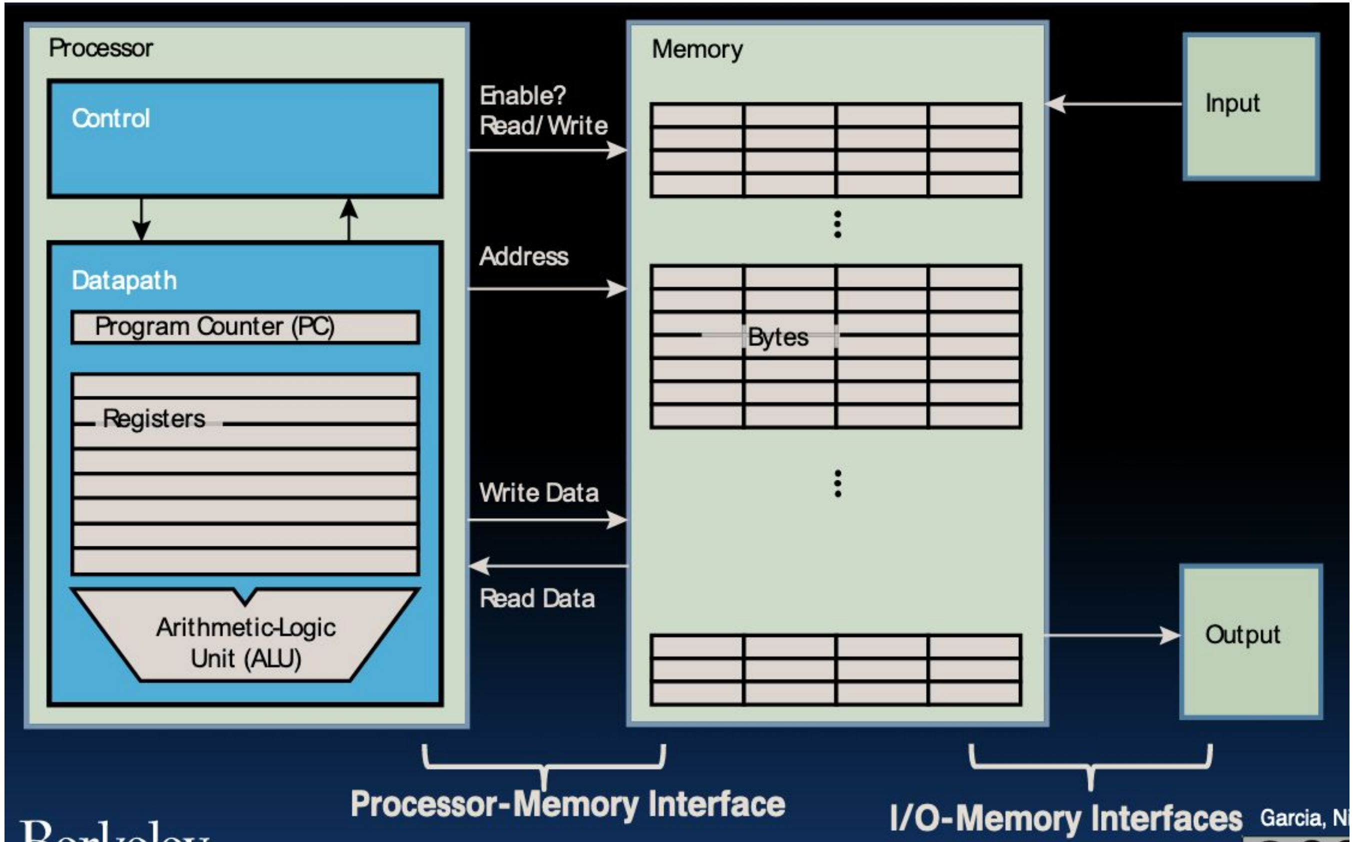
$X=0 \Rightarrow$	---	$Y=0 \Rightarrow$	1
$X=1 \Rightarrow$	kibi ~10^3	$Y=1 \Rightarrow$	2
$X=2 \Rightarrow$	mebi ~10^6	$Y=2 \Rightarrow$	4
$X=3 \Rightarrow$	gibi ~10^9	$Y=3 \Rightarrow$	8
$X=4 \Rightarrow$	tebi ~10^{12}	$Y=4 \Rightarrow$	16
$X=5 \Rightarrow$	pebi ~10^{15}	$Y=5 \Rightarrow$	32
$X=6 \Rightarrow$	exbi ~10^{18}	$Y=6 \Rightarrow$	64
$X=7 \Rightarrow$	zebi ~10^{21}	$Y=7 \Rightarrow$	128
$X=8 \Rightarrow$	yobi ~10^{24}	$Y=8 \Rightarrow$	256
		$Y=9 \Rightarrow$	512

MEMORIZE!

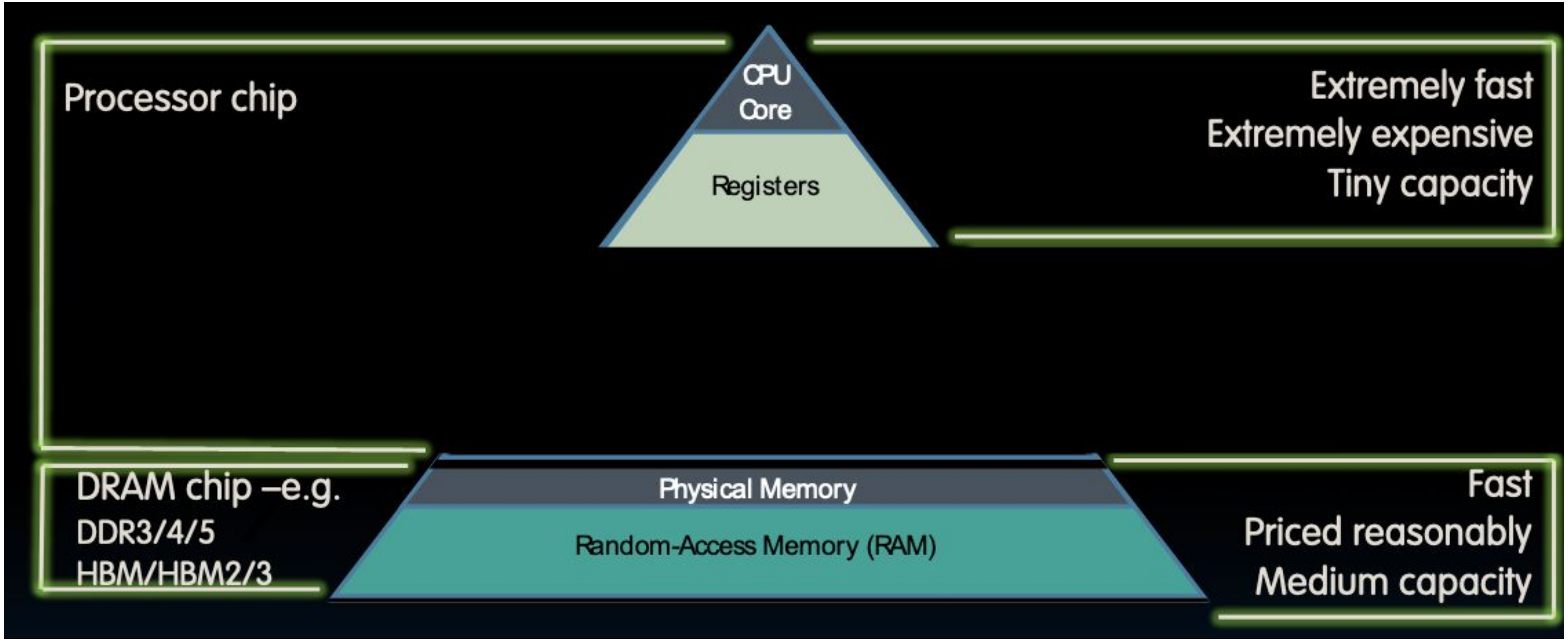
Review: Operation Time Costs

- As a general rule of thumb, the runtime cost of operations gets broken down as follows:
- File operations
  - Extremely slow (retrieving from disk, so it takes a long time)
  - Also includes prints and other I/O, so it's a good idea to avoid printing lots of data when testing runtime.
- Memory operations
  - ~100x faster than file operations (accessing RAM)
  - Can be improved through caching, but memory operations still take 10-100 clock cycles
- Branches and Jumps
  - Slower than most operations due to hazards (potentially 5 clock cycles)
  - Can be improved through loop unrolling
- Arithmetic operations
  - Fastest operations (1 cycle ideally, though some operations take longer)

Components of a Computer



Memory Hierarchy

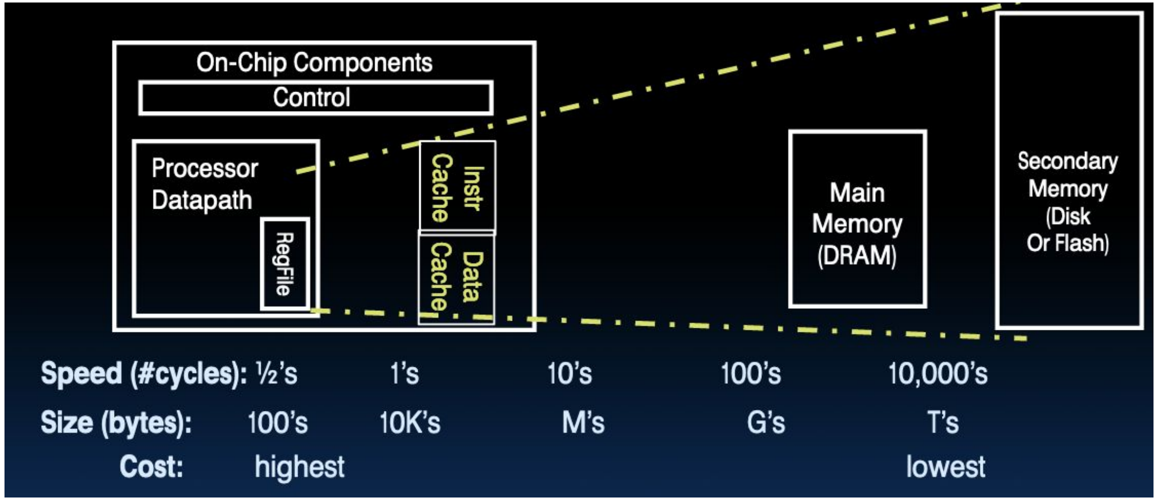


Memory Hierarchy

- Registers are fast to use and close to the CPU, but they're expensive (both in energy cost and in manufacturing costs), so we can only have a few of them per CPU
- DRAM is much better-suited for storing kibibytes of data, but ~100x slower to access
    - Different scale means we need to use different circuit components, so we can't just mux together a bunch of registers together
    - Need things like data buses to facilitate memory transfer
- Today's topic: Reducing the amount of time we spend waiting on memory operations

Typical Memory Hierarchy

- The Trick: present processor with as much memory as is available in the cheapest technology at the speed offered by the fastest technology


On-Chip Components
Control
Processor Datapath
RegFile
Instr Cache
Data Cache
Main Memory (DRAM)
Secondary Memory (Disk Or Flash)
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

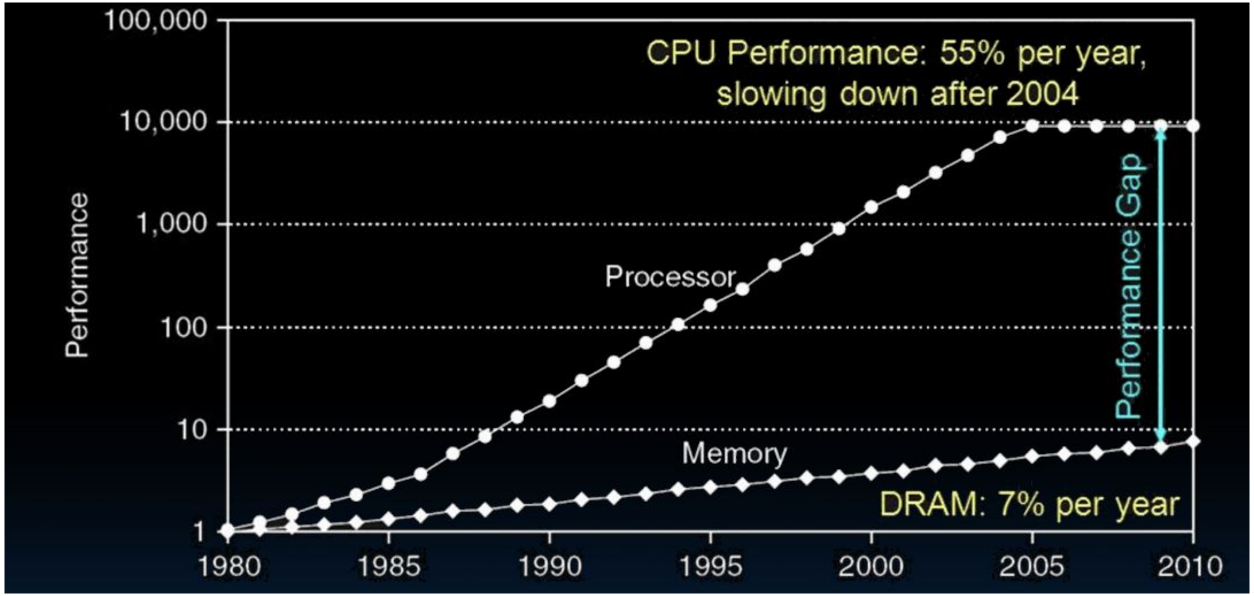
Why are large memories slow? Library Analogy

- You want to write a report using library books
    - E.g., works of J.D. Salinger
- Our desk at home is the CPU; we can only do work while at home
- The library is the DMEM; it contains all the information we need to write our report
    - Also, notebooks are also stuck at the library, so we store all our heap memory in the library
- Currently: When we need to read a memory address, we call the library and ask for them to find our data.
- The librarians search the library, finds the particular book, and finds the line within the book we need.
- The librarians memorize the line, put the book back, and call us back with the line of data we need

Why are large memories slow? Library Analogy

- Time to find a book in a large library
    - Search a large card catalog – (mapping title/author to index number)
    - Round-trip time to walk to the stacks and retrieve the desired book
- Larger libraries worsen both delays
- Electronic memories have same issue, plus the technologies used to store data slow down as density increases (e.g., SRAM vs. DRAM vs. Disk)

Processor-DRAM Gap (Latency)



1980 microprocessor executes ~one instruction in same time as DRAM access
2020 microprocessor executes ~1000 instructions in same time as DRAM access
Slow DRAM access has disastrous impact on CPU performance!

Library Analogy: Ways to speed things up?

- Option 1: While you're waiting for the data you need to do your English paper, switch to a different task and do your math homework
- Option 2: Plan ahead and ask for the data ahead of time. By the time you need that data, the librarians are ready to call you back
- Option 3: Instead of just getting one line of data at a time, borrow some of the library books and store them at home so you don't need to go all the way to the library

Option 1: Hyperthreading

- Option 1: While you're waiting for the data you need to do your English paper, switch to a different task and do your math homework
- In a computer: While waiting for a DMEM access, perform a context switch and move to another thread
    - Context switching is expensive, so needs specialized hardware to get speedup
- Each physical core can contain two "virtual cores" with their own set of registers, so context switching is faster
    - Ex. My computer has 12 cores, 16 processors
- Intel's version of this technology is called hyperthreading
- Done at the OS level, and doesn't speed up an individual task

Option 2: Prefetching

- Option 2: Plan ahead and ask for the data ahead of time. By the time you need that data, the librarians are ready to call you back
- In a computer: Request data several cycles before you actually use the data
  - Similar to reordering operations to reduce hazards
- Can be done manually with prefetching commands in C, but this is out of scope for this course.
- Software-based, so flexible, but also limited.

Option 3: Caching

- Option 3: Instead of just getting one line of data at a time, borrow some of the library books and store them at home so you don't need to go all the way to the library
- Main idea: If you're writing a report on Salinger, there's a good chance that you know what books you'll need
- If you use a book once, chances are you'll use it again.
- Solution: Get a bookshelf at home that you can use to keep a few books, and "borrow" the book from the library when you read it.
- If we need another book, go back to the library and borrow a new one
  - But don't return earlier books since you might need them, until your bookshelf is too full to carry more books
- You hope this collection of ~10 books at home enough to write report, despite 10 being only 0.00001% of books in UC Berkeley libraries

Caching

- Caching as a whole is the idea of saving data we'll need soon somewhere close by, so we can access it sooner.
- Has applications in many fields
  - Dynamic programming is basically a form of caching
  - Helps avoid lag spikes in video streaming (ex. Youtube)
- Today, we'll focus on data caches, which are hardware components that store 1 KiB-1 MiB of memory.
- Goal: store a subset of main memory that is most useful to keep
  - Caching ends up adding a small overhead to all memory accesses, but on average speeds things up if we're good at predicting useful memory.
- Since it's a hardware component, we can't change our cache structure based on the program, so we need to find a way to predict this for most programs.

Locality

- Main idea: predict what memory might be accessed next based on what memory was already accessed
- Temporal Locality
  - If a memory location is referenced then it will tend to be referenced again soon
    - Ex. Matmul reads the same matrix multiple times in quick succession
  - Keep most recently accessed data items in our cache
- Spatial Locality
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
    - Ex. Matmul accesses memory in a contiguous sequence of addresses
  - Move blocks consisting of contiguous words instead of just one word at a time

Blocks

- Move blocks consisting of contiguous words instead of just one word at a time
- To facilitate this, we will divide all memory into "blocks" of size some power of 2
  - Analogy: Each memory address is a line in a book, each block is a book
- Blocks will be assigned a number, called their tag
- Ex. If we have blocks of size 4096 = 2^{12}:
  - Block `0` will contain data from `0x0000 0000` to `0x0000 0FFF`
  - Block `1` will contain data from `0x0000 1000` to `0x0000 1FFF`
  - Block `0xABCDE` will contain data from `0xABCD E000` to `0xABCD EFFF`
- A memory address can be divided into a tag, and an offset
  - Ex. `0xDEADBEEF` with a block size of 4096
  - This address is in Block `0xDEADB`, and is the `0xEEF`th byte in that block

Memory Hierarchy Basis

- Caches are an intermediate memory level between fastest and most expensive memory (registers) and the slower components (DRAM)
  - DRAM can be considered a "cache" for disk in a way!
- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Caches work on the principles of temporal and spatial locality.
  - Temporal locality (locality in time): If we use it now, chances are we'll want to use it again soon.
  - Spatial locality (locality in space): If we use a piece of memory, chances are we'll use the neighboring pieces soon.

How is the hierarchy managed?

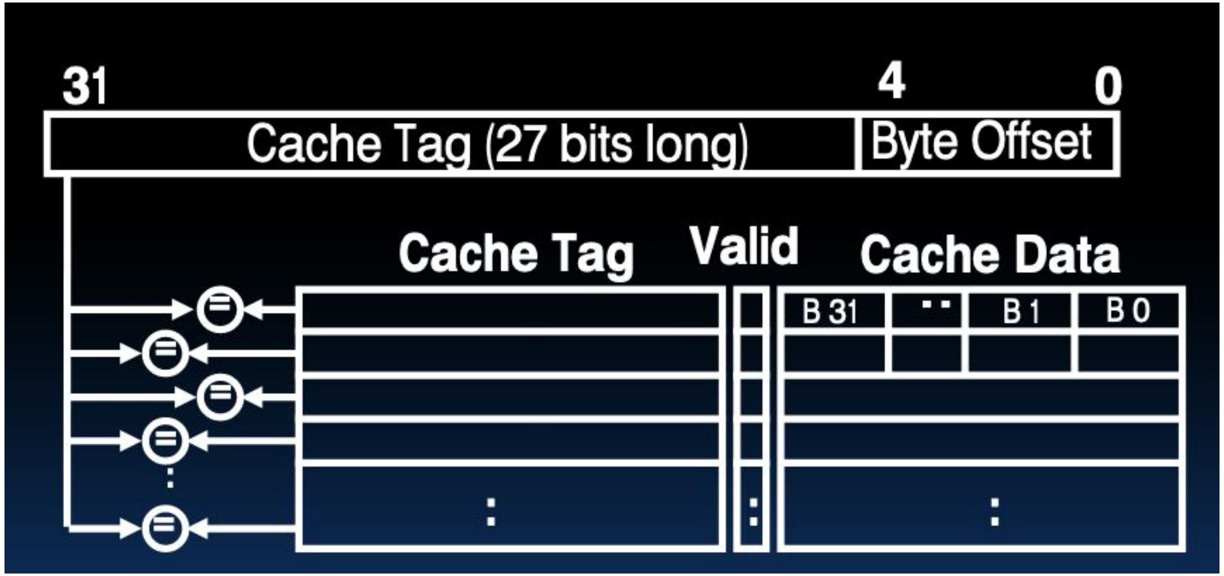
- registers « memory
  - By compiler (or assembly level programmer)
- cache « main memory
  - By the cache controller hardware
- main memory « disks (secondary storage)
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware (‘translation lookaside buffer’ or TLB)
  - By the programmer (files)

Fully Associative Cache (1/4)

- A fully associative cache is parametrized on two aspects: The block size, and the number of blocks that can be stored in the cache.
- When a memory access occurs:
  - Step 1: Check if the cache contains the data needed. If so, return the data.
  - Step 2: If the data is not present in the cache, load the block of memory into the cache
  - Step 3: Return the data
- Library analogy: You buy a bookshelf that can hold N books
  - Step 1: Check if your bookshelf has the book you need. If so, get the data
  - Step 2: If you don't have the needed book, go to the library to borrow it
  - Step 3: Return the data.

Fully Associative Cache (2/4)

- Fully Associative Cache (with 32 B block)
  - Tag identifies what block is stored
  - Valid is a 1-bit value that differentiates actual data from garbage
    - In any hardware system, each bit is always set to something.



Fully Associative Cache (3/4): Example Memory Access

- Let's say we have a cache with 4 byte blocks and 4 blocks storage, and let's assume that we're working with a system that uses 10-bit addresses
- To access address 0x3C1:
  - Split into tag and offset:
    - `0x3C1 == 0b1111 0000 01`
  - Tag matches a block in our cache, and the valid bit is on, so our data is in the cache
  - We get the 1st byte of the block, which is `0xDE`

Tag	Valid	Data
0b1001 1001	0	0xDE 0xAD 0xBE 0xEF
0b1011 0011	0	0x01 0x23 0x45 0x67
0b0110 1011	1	0xAB 0xAD 0xCA 0xFE
0b1111 0000	1	0xAC 0xDE 0xFF 0x61

Fully Associative Cache (3/4): Example Memory Access

- Let's say we have a cache with 4 byte blocks and 4 blocks storage, and let's assume that we're working with a system that uses 10-bit addresses
- To access address 0x266:
  - Split into tag and offset:
    - `0x266 == 0b1001 1001 10`
  - Tag matches a block in our cache, but the valid bit is off, so our data is not in our cache
  - We need to go to main memory to load the block.

Tag	Valid	Data
`0b1001 1001`	`0`	`0xDE 0xAD 0xBE 0xEF`
`0b1011 0011`	`0`	`0x01 0x23 0x45 0x67`
`0b0110 1011`	`1`	`0xAB 0xAD 0xCA 0xFE`
`0b1111 0000`	`1`	`0xAC 0xDE 0xFF 0x61`

Fully Associative Cache (4/4): Cache Terminology

- When reading memory, 3 things can happen:
  - **Cache Hit**:
    Cache block is valid and contains proper address, so read desired word
  - **Cache Miss**:
    Nothing in cache in appropriate block, so fetch from memory and replace an invalid block
  - Cache Miss with **Eviction**:
    Cache Miss, but the cache is already full of valid data, so we need to remove one block from the cache before we can load the new block
    - How to choose which block to evict is known as the **Eviction Policy** - more on this soon
- A cache is considered **Cold** if it doesn't have valid data in it
  - Often when starting a computer initially, or when recovering from a context switch
- A cache is considered **Hot** if it has valid data, and there's a high ratio of Cache Hits

Cache Timing

- Compare the naive approach (just going to the library) to a cache. With a cache:
  - We need to check our cache for data, so this takes a small amount of runtime
  - There's a chance that we don't need to go to the library at all, because our data is in the cache, so this saves a lot of runtime **sometimes**. The hotter the cache, the more likely we save runtime.
- Since the cache is a hardware component, we are forced to use it for all programs
  - Therefore, if our program isn't cache-efficient, this can slow down runtime!
  - No free lunch theorem: For a random sequence of memory operations, a cache will on average slow down total runtime, unless the cache contains all data in main memory.
  - We get speedup only because we never use truly random memory accesses
- In practice, cache hits are around 10 cycles, while cache misses are around 100-1000 cycles, so we get a big improvement from every cache hit.

Cache Variants

- What we've discussed today is a **fully-associative cache.**
- Not ideal for all purposes; if we have N blocks, we need N comparators, and that's expensive
- Wednesday: Different types of caches that sacrifice hit rate for hit time and hit speed