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

Lecture 18
Caches I: Intro to Caches, Fully Associative Caches

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

Announcements

- Exam prep sections continue
    - Parallelism and some caches: Thursday 1-3pm in Soda 380, Friday 1-3pm in Cory 540AB
- Project 3 part A due tonight!
- HW5 due tomorrow, Wednesday 7/26
- Labs 7 and 8 due Thursday 7/27

Agenda

- Review: Binary Prefixes
- Why Caches?
- Intro to Caches
- Fully Associative Cache
- Replacement and Write Policies

Review: Binary Prefixes

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

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

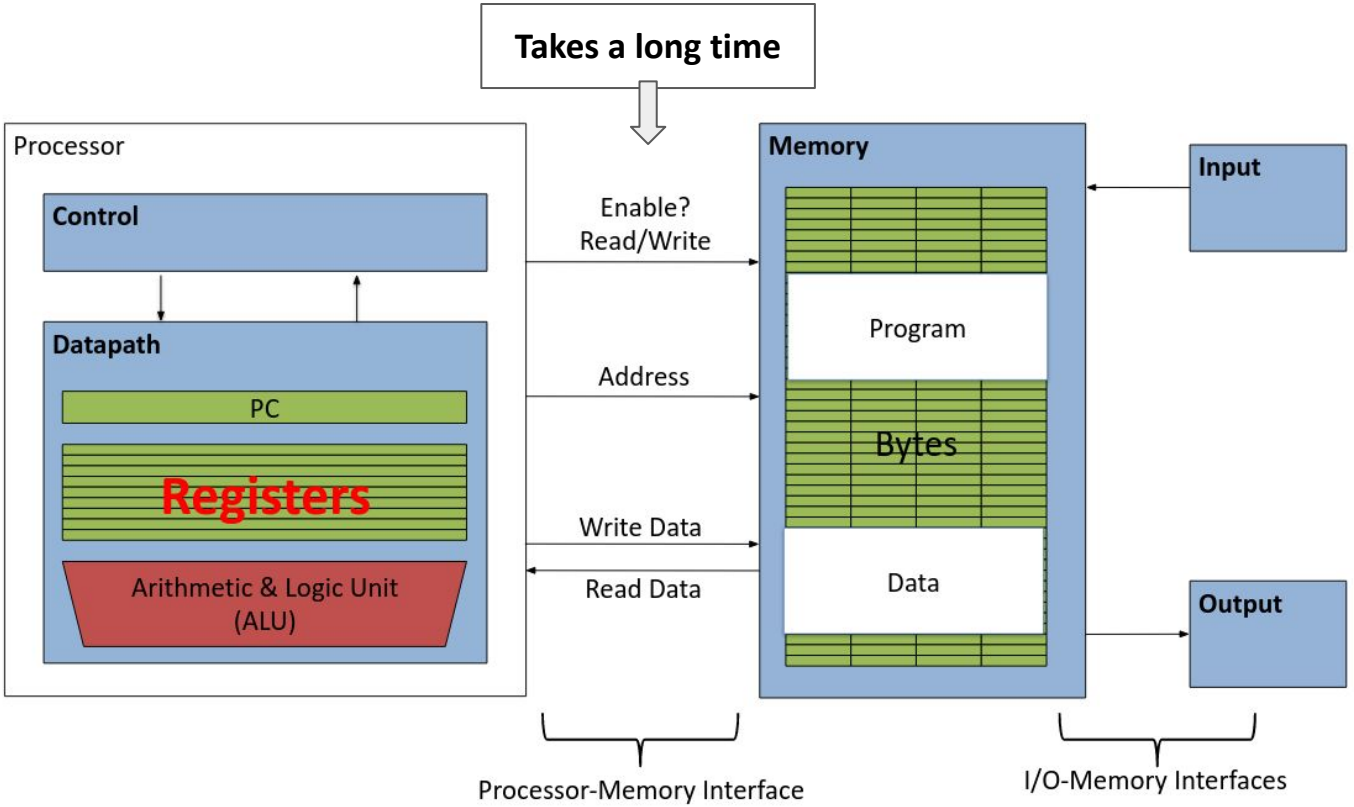
- International Electrotechnical Commission (IEC) in 1999 introduced these to specify binary quantities.
- Names come from shortened versions of the original SI prefixes (same pronunciation) and bi is short for “binary”, but pronounced “bee” :-(
- Now SI prefixes only have their base-10 meaning and never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

Why Caches?

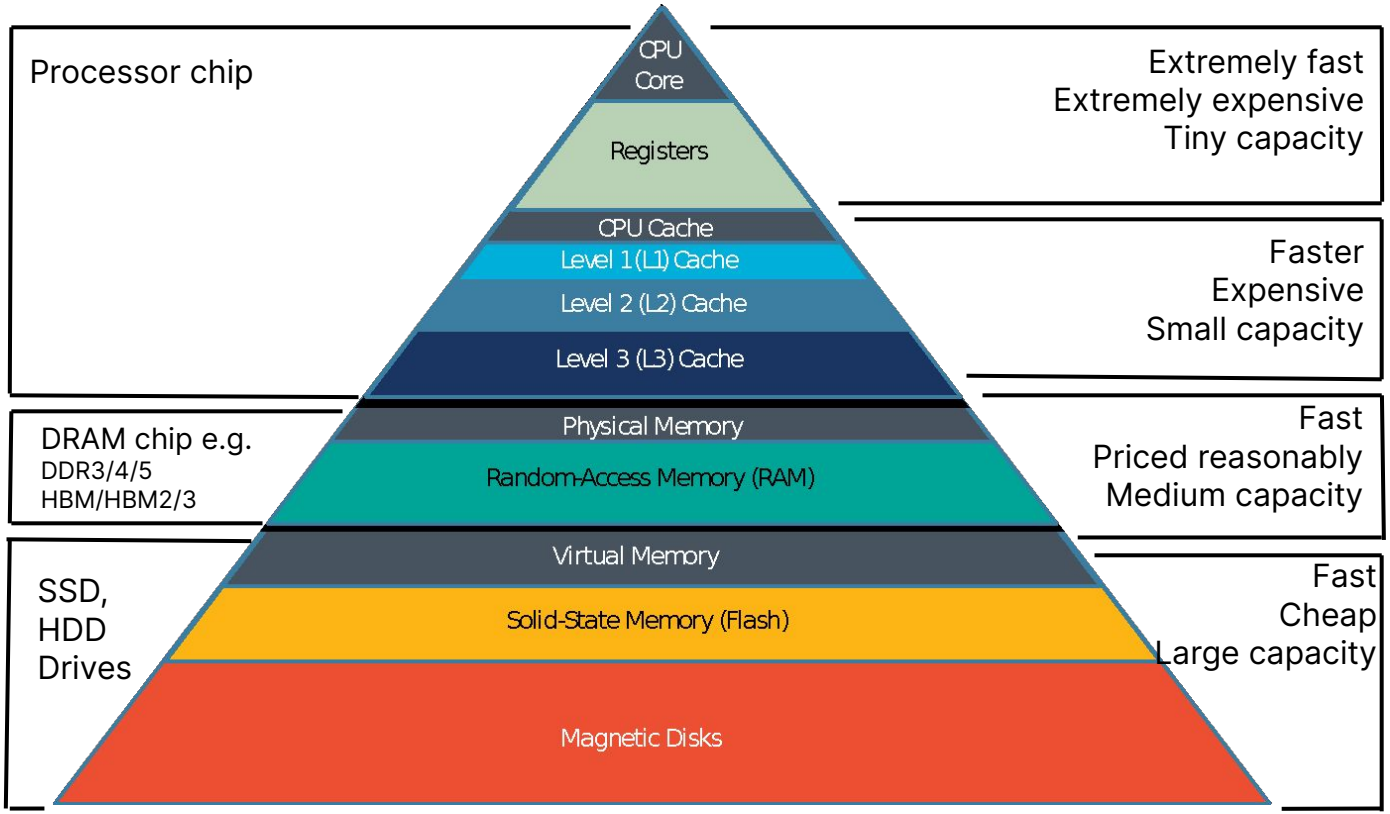
Review: Operation Time Costs

- Arithmetic operations
  - Fastest operations (1 cycle ideally, though some operations take longer)
- Branches and Jumps
  - Slower than most operations due to hazards (potentially 5 clock cycles)
  - Can be improved through loop unrolling
- Memory operations
  - Take 100-200 clock cycles
  - Can be improved through caching
- File operations
  - Extremely slow (retrieving from disk, so it takes a long time) – ~100x slower than memory
  - Similar: prints and other I/O

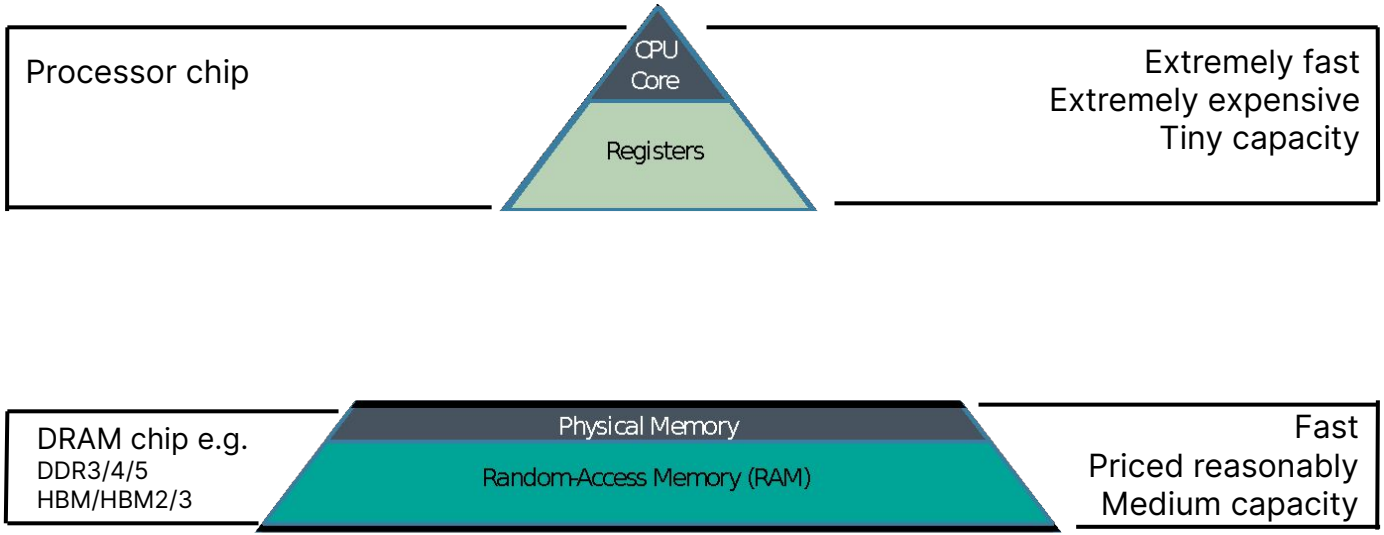
Components of a Computer



Great Idea #3: Principles of Locality / Memory Hierarchy



Great Idea #3: Principles of Locality / Memory Hierarchy

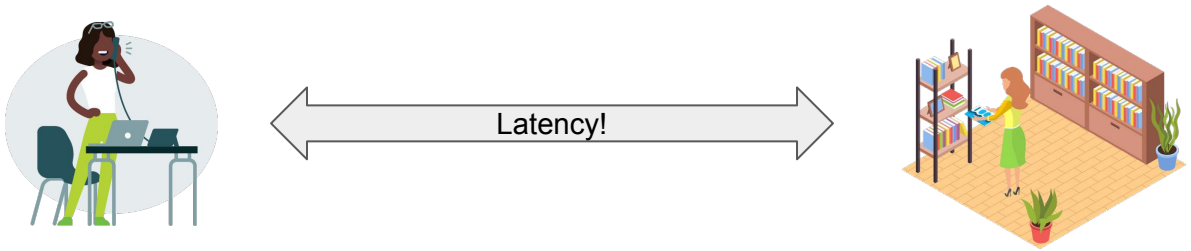


Memory Hierarchy

- 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 or more of data, but ~100x slower to access
  - Can't just mux together thousands of registers
  - Need things like data buses to facilitate memory transfer
- Today's topic: Reducing the amount of time we spend waiting on memory operations

Library Analogy (1/2)

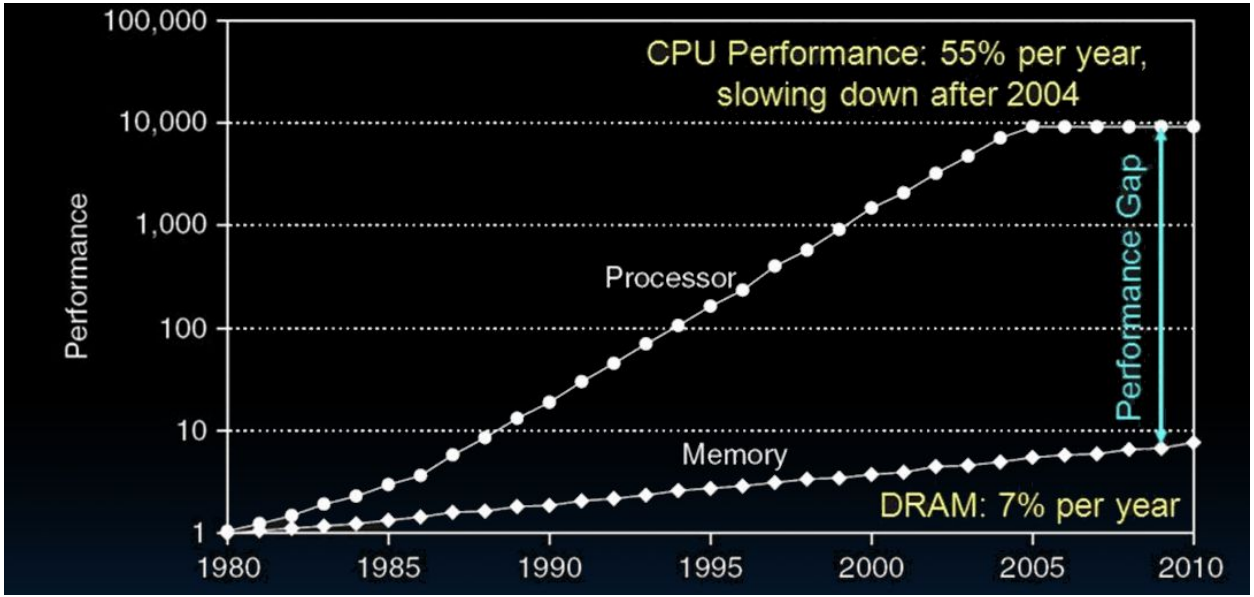
- You want to write a report using library books
  - E.g., works of J.D. Salinger
- Our desk at home is the CPU; we can only do work while at home
- The library is the DMEM; it contains all the information we need to write our report
- Currently: When we need to read a memory address, we call the library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



Library Analogy (2/2)

- The larger the library the longer it takes to find information
  - Search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book
- 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)



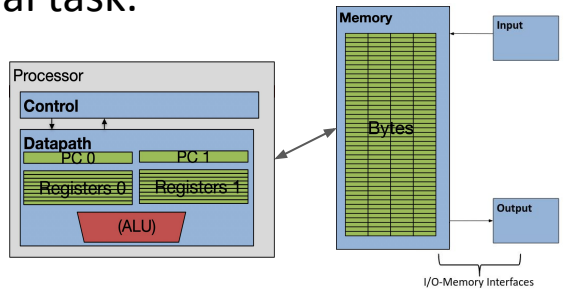
1980 CPU executes ~1 instruction in same time as DRAM access
2020 CPU executes ~1000 instructions in same time as DRAM access
Slow DRAM access has disastrous impact on CPU performance!

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: Hardware Multithreading

- 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
- Recall: Each physical core can contain two sets of PCs and registers, so context switching is faster
- Done at the OS level, and doesn't speed up an individual task.



Option 2: Prefetching

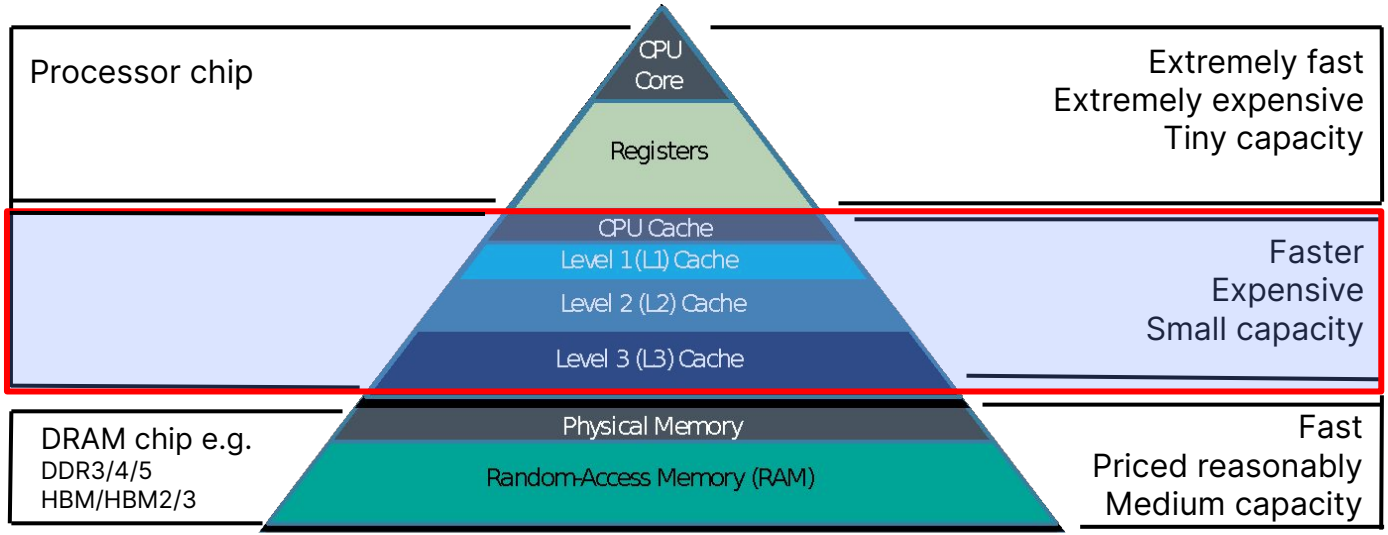
- Option 2: Plan ahead and ask for the data ahead of time. By the time you need that data, the librarians are ready to call you back
- In a computer: Request data several cycles before you actually use the data
- Can be done manually with prefetching commands in C (limited effectiveness)
- Software-based, 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
  - Don’t return unless your bookshelf is too full to carry more books, since we might need them later
- You hope this collection of ~10 books at home enough to write report, despite being only 0.00001% of books in UC Berkeley libraries.

Great Idea #3: Principles of Locality / Memory Hierarchy

Caches!



Caching

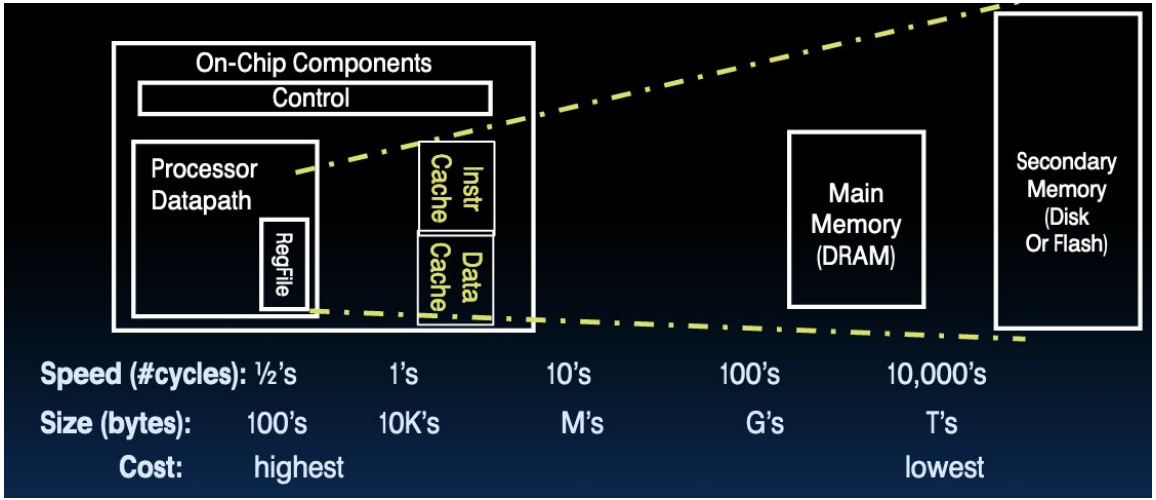
- **Caching**: save data we're likely to use somewhere close by, so we can access it sooner.
    - Has applications in many fields, e.g. the Internet would be orders of magnitude slower without caches
- Today, we'll focus on CPU caches, which are hardware components that store approx. a few KiB to a few MiB of memory.
- Goal: store a subset of main memory that is most useful to keep

Memory Hierarchy Basis

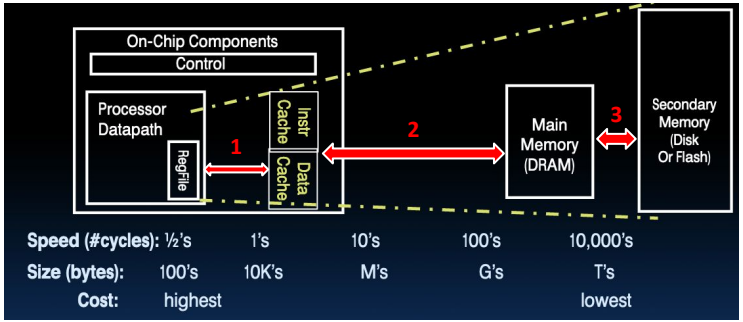
- Caches are an intermediate memory level between fastest and most expensive memory (registers) and the slower components (DRAM)
- **Cache** contains copies of data in **memory** that are being used.
- **Memory** contains copies of data on **disk** that are being used.
  - Memory can be considered a “cache” for the disk
  - More: next unit (virtual memory)!

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



How is the hierarchy managed?



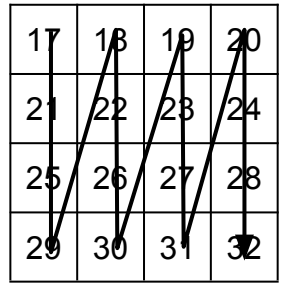
1. Registers ↔ Memory
   - By **compiler** (or assembly level programmer)
2. Cache ↔ Main memory
   - By the **cache controller hardware**
3. Main memory ↔ Disks (secondary storage)
   - By the **operating system** (virtual memory)
   - Virtual to physical address mapping assisted by the hardware ('translation lookaside buffer' or TLB)
   - By the programmer (files)

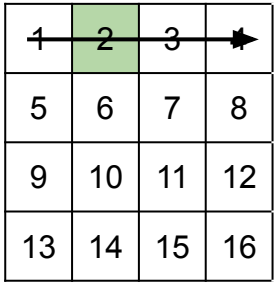
Locality (1/3)

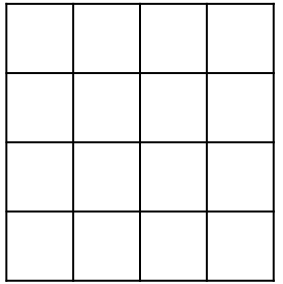
- Main idea: predict what memory might be accessed next based on what memory was already accessed
- **Temporal Locality**
  - Means: If a memory location is referenced then it will tend to be referenced again soon
  - Consequence: Keep most recently accessed data items in our cache
- **Spatial Locality**
  - Means: If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
  - Consequence: Move blocks consisting of contiguous words instead of just one word at a time

Locality (2/3)

- **Temporal Locality**
  - If a memory location is referenced then it will tend to be referenced again soon
  - Example: In a matrix multiplication, we access the same element multiple times in sequence, to multiply it to each column.

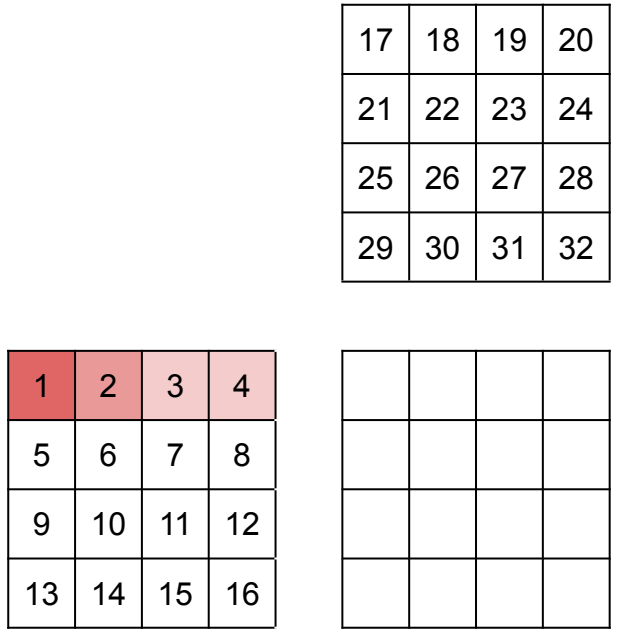






Locality (3/3)

- **Temporal Locality**
  - If a memory location is referenced then it will tend to be referenced again soon
  - Example: In a matrix multiplication, we access the same element multiple times in sequence, to multiply it to each column
- **Spatial Locality**
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
  - Example: In a matrix multiplication, when we access one element, we also access nearby elements at a similar time.



Blocks

- Consequence of spatial locality:
  - 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.

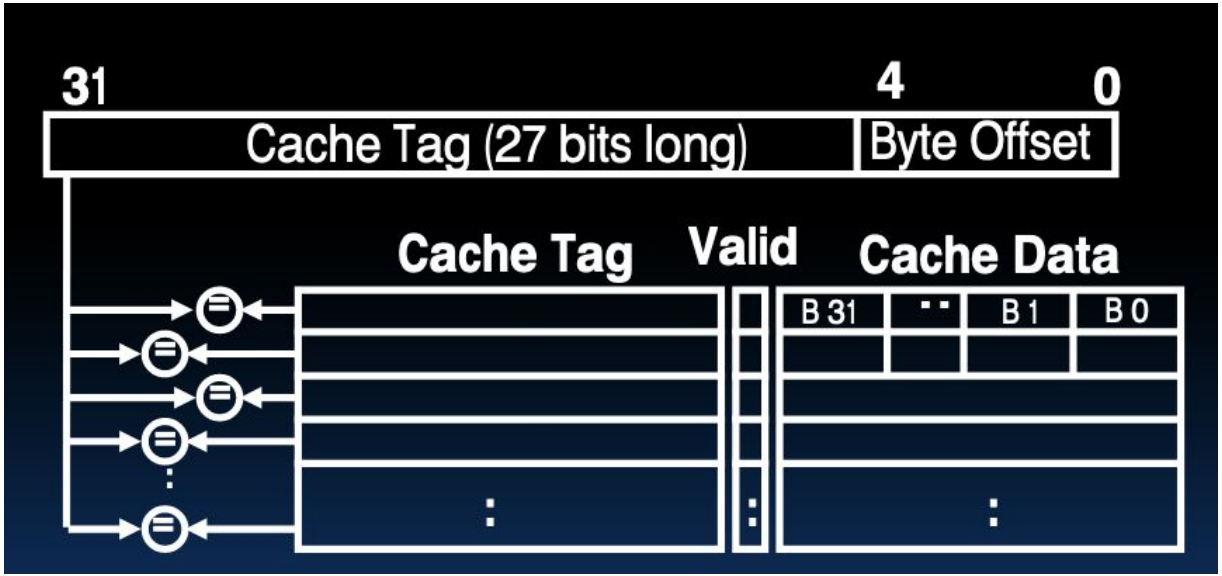
Fully Associative Cache

Fully Associative Cache (1/2)

- 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
- A fully associative cache is parameterized on two aspects: The block size, and the number of blocks that can be stored in the cache.

Fully Associative Cache (2/2)

- 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: Example Memory Access (1/2)

- Let's say we have a cache with 4 byte blocks and 4 blocks storage
- Addresses are 10 bits
- 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`

Tag: 0b11110000

Offset: 1

Fully Associative Cache: Example Memory Access (2/2)

- 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`

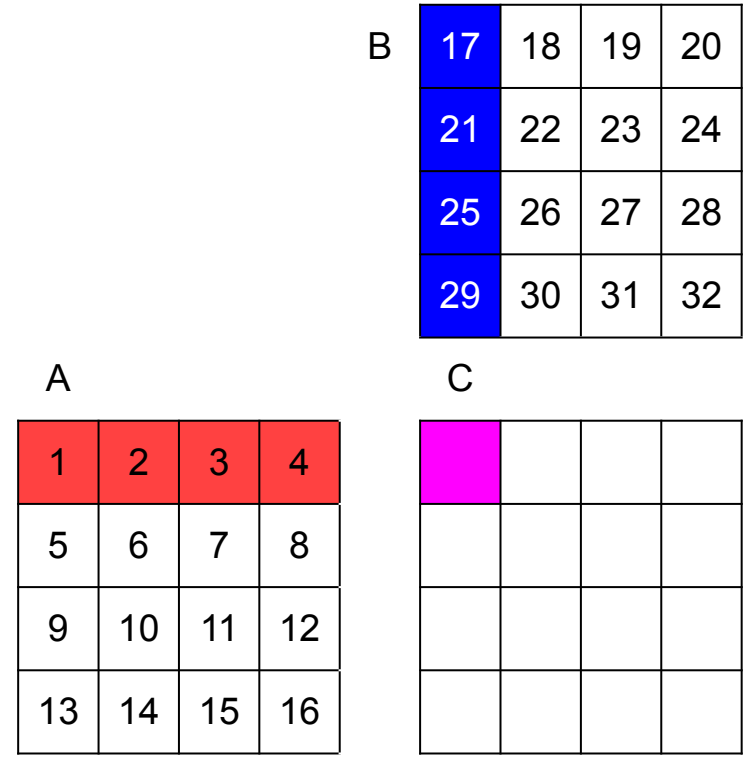
Fully Associative Cache: Cache Terminology

- 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 **Replacement 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

Replacement and Write Policies

Caching Example: Matrix Multiply

- Let’s use Matrix Multiply as an example again.
- Assumptions:
  - Elements are 4 bytes
    - But only the bottom byte will be shown in the cache for space
  - Block size is 16 bytes (one row of this matrix)
  - Matrix A stored at 0x1000
  - Matrix B stored at 0x2000
  - Matrix C stored at 0x3000
  - No other memory used
  - Caches start cold
- Let's see how this works on a fully associative cache with 4 blocks



Caching Example: Fully Associative

- Start: Cache starts cold
  - Note: random data in the cache, but all valid bits are zero
- 0 misses, 0 hits

Tag	Valid	Data
0x100	0	0xBF 0x16 0x88 0x2B
0x733	0	0x3B 0x18 0xF1 0xB3
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

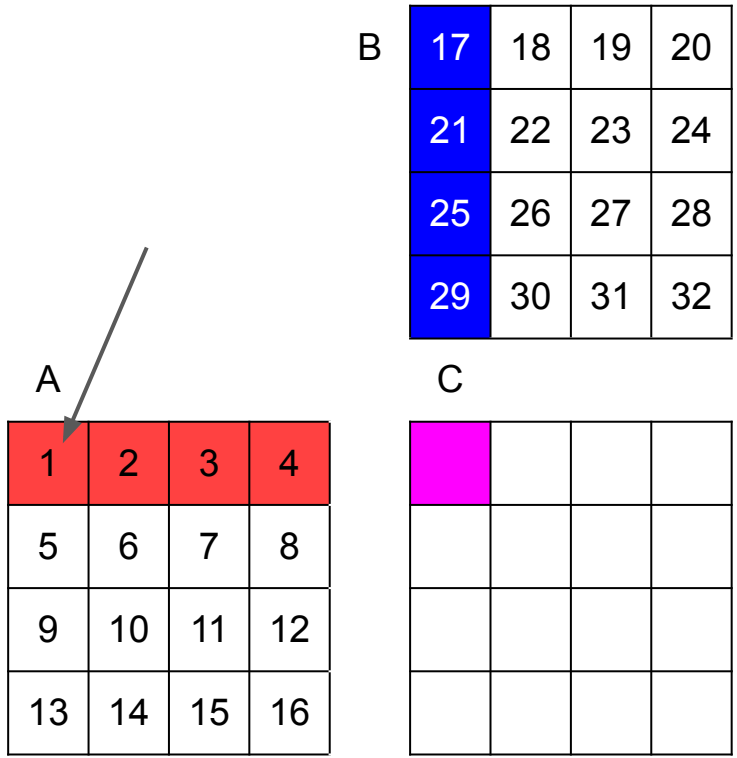
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

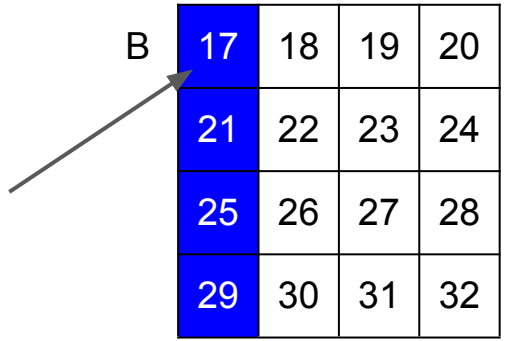
- Access A[0] = 0x1000
  - Miss: Data isn't valid, so pull from main memory
- 1 miss, 0 hits

Tag	Valid	Data
0x100	**0**	**0xBF 0x16 0x88 0x2B**
0x733	**0**	**0x3B 0x18 0xF1 0xB3**
0x156	**0**	**0xE6 0x57 0x49 0xEE**
0x4E9	**0**	**0xB5 0x81 0x67 0x3F**



Caching Example: Fully Associative

- Access B[0] = 0x2000
  - Miss: Data isn't valid, so pull from main memory
- 2 miss, 0 hits



Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x733	0	0x3B 0x18 0xF1 0xB3
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F

A

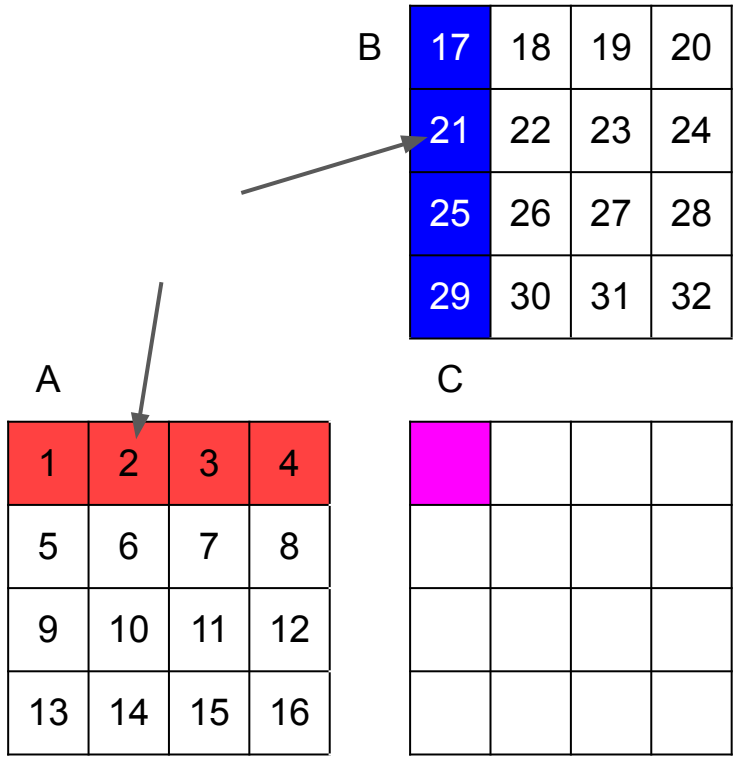
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Access A[1]: 0x1004
  - Hit
- Access B[4]: 0x2010
  - Miss
- 3 Misses, 1 Hit

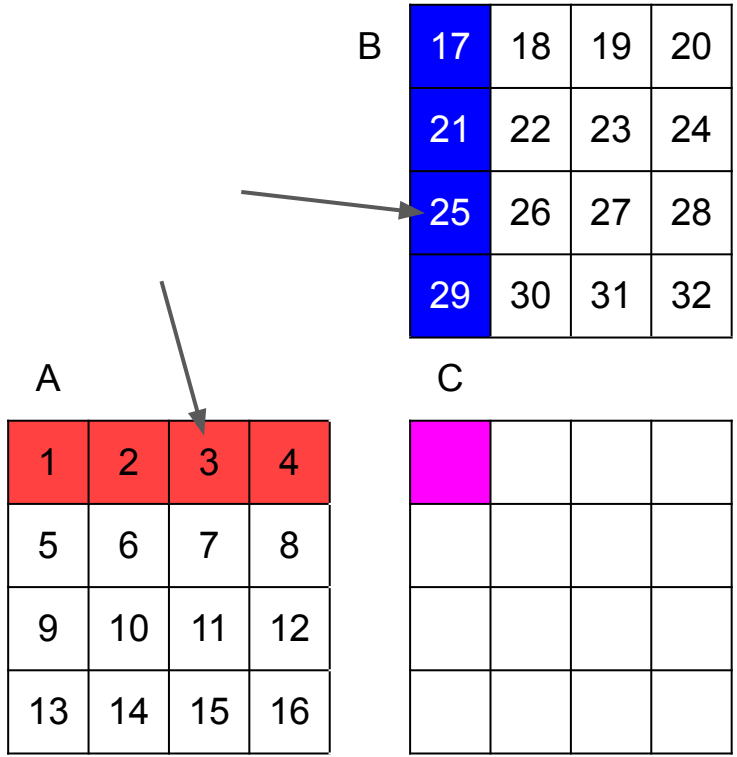
Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x200	1	0x11 0x12 0x13 0x14
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F



Caching Example: Fully Associative

- Access A[2]: 0x1008
  - Hit
- Access B[8]: 0x2020
  - Miss
- 4 Misses, 2 Hit

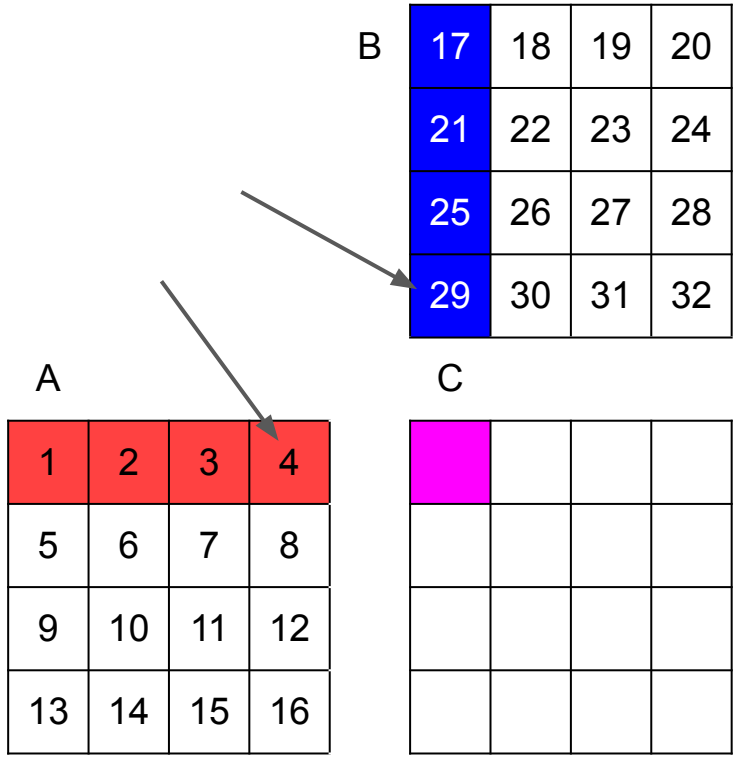
Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x200	1	0x11 0x12 0x13 0x14
0x201	1	0x15 0x16 0x17 0x18
0x4E9	0	0xB5 0x81 0x67 0x3F



Caching Example: Fully Associative

- Access A[3]: 0x100C
  - Hit
- Access B[12]: 0x2030
  - Miss
- Problem: The cache is full
  - Which block do we evict to make space?

Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x200	1	0x11 0x12 0x13 0x14
0x201	1	0x15 0x16 0x17 0x18
0x202	1	0x19 0x1A 0x1B 0x1C



Replacement Policies

- Our cache is necessarily smaller than main memory, so we will eventually reach a point where our cache is full, and we need to evict some block.
- Ideally, choose a replacement policy that evicts the least useful data. We have several options:
  - LRU
  - MRU
  - FIFO
  - LIFO
  - Random

Replacement Policies

- Least Recently Used (LRU): evict the block in the set that is the oldest previous access.
  - Pro: Takes full advantage of temporal locality
  - Con: Requires complicated hardware and much time to keep track of this
- Most Recently Used (MRU): evict the block in the set that is the newest previous access.
  - Counterintuitive, but would actually work better for some programs!
- First-In-First-Out (FIFO): evict the oldest block in the set (queue).
- Last-In-First-Out (LIFO): evict the newest block in the set (stack).
  - Both FIFO and LIFO ignore order of accesses after/before the first/last access, but they serve as good approximations to LRU and MRU without adding too much excess hardware
- Random: randomly selects a block to evict
  - Works surprisingly okay, especially given a low temporal locality workload

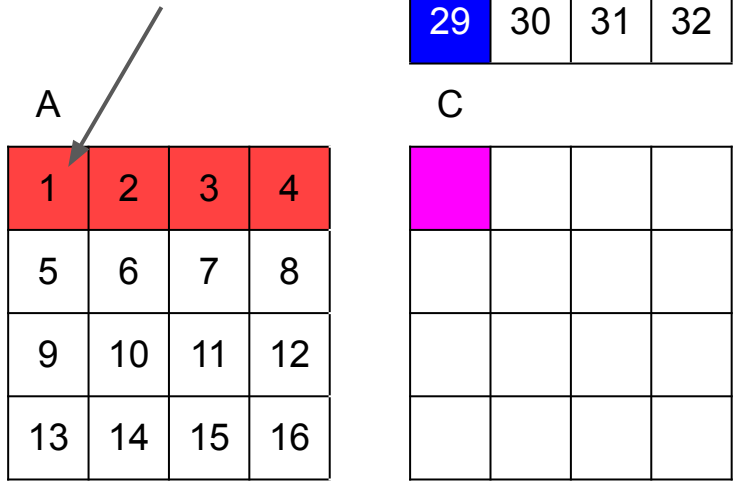
Replacement Policies

- Our cache is necessarily smaller than main memory, so we will eventually reach a point where our cache is full, and we need to evict some block.
- Ideally, choose a replacement policy that evicts the least useful data. We have several options
    - Impossible to select the *very best* policy without knowing program ahead of time
- For now: Let's use an LRU cache

Caching Example: Fully Associative with LRU

- Start: Cache starts cold
  - Note: New LRU field containing which block accessed first
    - Lower number = more recent for today
- 0 misses, 0 hits

Tag	Valid	**LRU**	Data
0x100	0	0	**0xBF 0x16 0x88 0x2B**
0x733	0	0	**0x3B 0x18 0xF1 0xB3**
0x156	0	0	**0xE6 0x57 0x49 0xEE**
0x4E9	0	0	**0xB5 0x81 0x67 0x3F**



Caching Example: Fully Associative with LRU

- Access A[0] = 0x1000
  - Miss

Tag	Valid	LRU	Data
0x100	0	0	**0xBF 0x16 0x88 0x2B**
0x733	0	0	**0x3B 0x18 0xF1 0xB3**
0x156	0	0	**0xE6 0x57 0x49 0xEE**
0x4E9	0	0	**0xB5 0x81 0x67 0x3F**

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

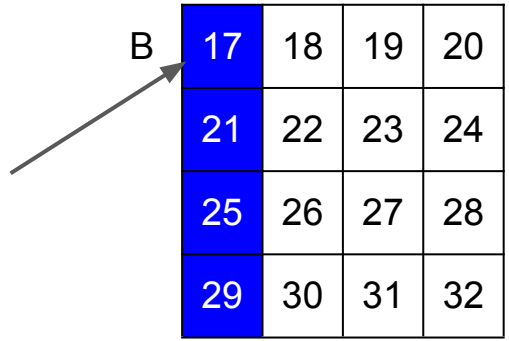
A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

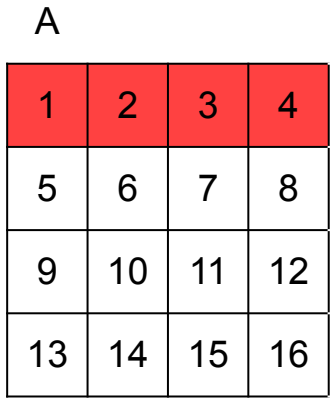
C

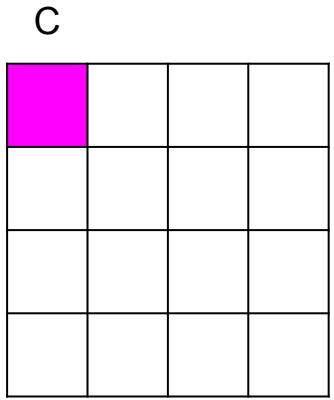
Caching Example: Fully Associative with LRU

- Access B[0] = 0x2000
  - Miss



Tag	Valid	LRU	Data
0x100	1	1	0x01 0x02 0x03 0x04
0x733	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0xB5 0x81 0x67 0x3F

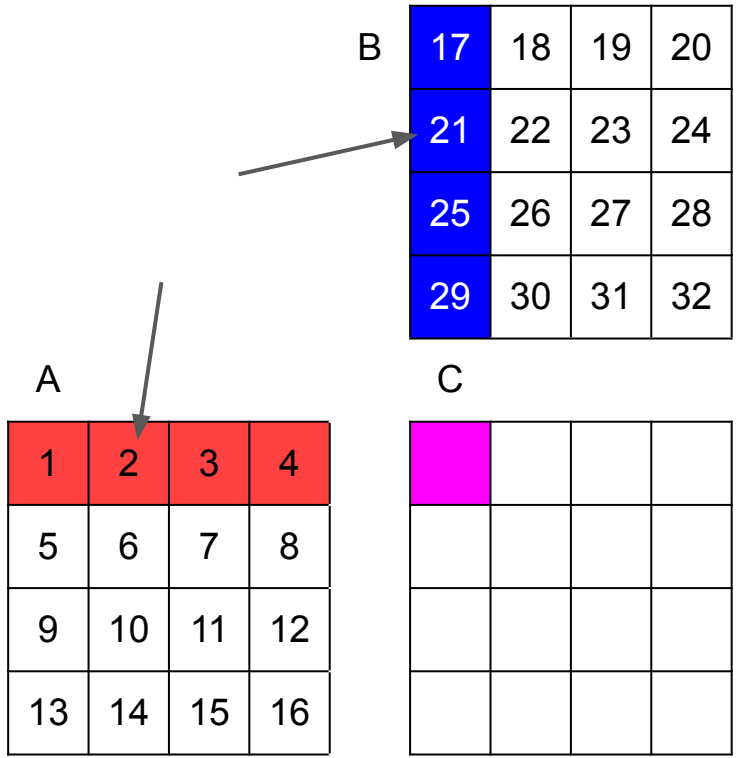




Caching Example: Fully Associative with LRU

- Access A[1]: 0x1004
  - Hit
- Access B[4]: 0x2010
  - Miss

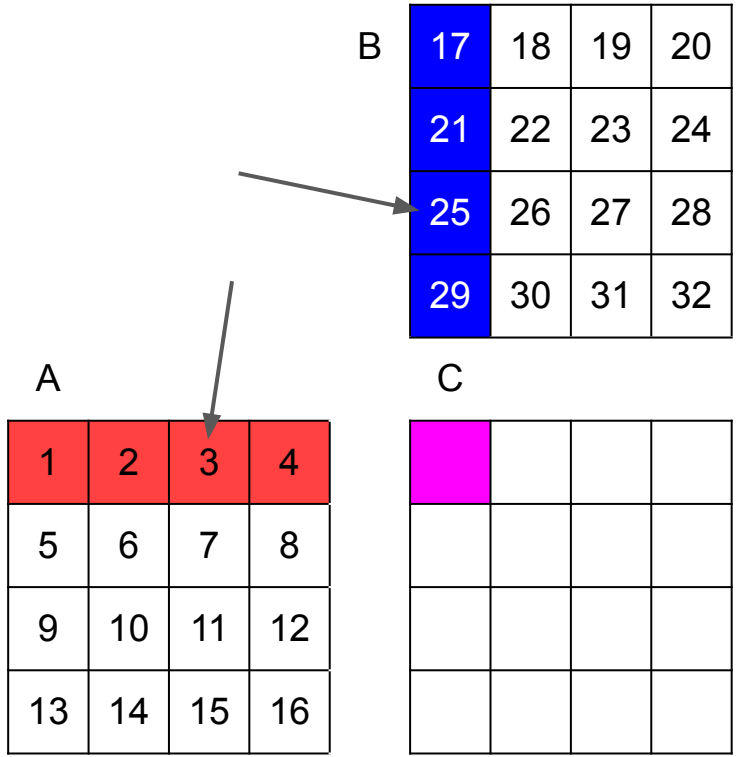
Tag	Valid	LRU	Data
0x100	1	2	0x01 0x02 0x03 0x04
0x200	1	1	0x11 0x12 0x13 0x14
0x156	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0xB5 0x81 0x67 0x3F



Caching Example: Fully Associative with LRU

- Access A[2]: 0x1008
  - Hit
- Access B[8]: 0x2020
  - Miss

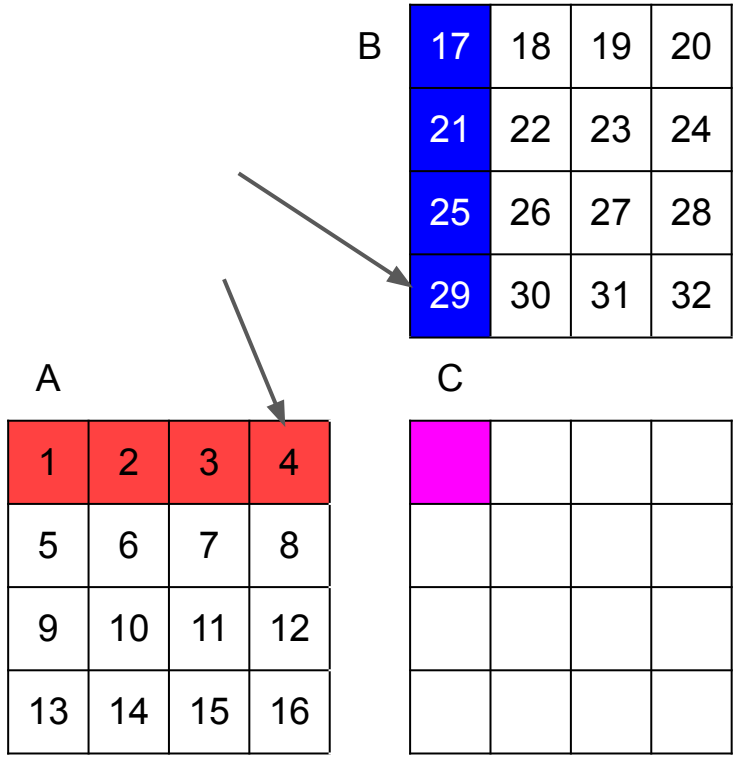
Tag	Valid	LRU	Data
0x100	1	2	0x01 0x02 0x03 0x04
0x200	1	3	0x11 0x12 0x13 0x14
0x201	1	1	0x15 0x16 0x17 0x18
0x4E9	0	0	0xB5 0x81 0x67 0x3F



Caching Example: Fully Associative with LRU

- Access A[3]: 0x100C
  - Hit
- Access B[12]: 0x2030
  - Miss
  - Cache is full, so evict the block with tag 0x200

Tag	Valid	LRU	Data
0x100	1	2	0x01 0x02 0x03 0x04
0x200	1	4	0x11 0x12 0x13 0x14
0x201	1	3	0x15 0x16 0x17 0x18
0x202	1	1	0x19 0x1A 0x1B 0x1C



Caching Example: Fully Associative with LRU

- Note: At this point, we've gotten a lot of misses, especially since we just evicted 0x2004, even though we never used it.
  - If we keep going, 60 hits and 68 misses from A and B alone
  - Transpose B to get more hits

Tag	Valid	LRU	Data
0x100	1	3	0x01 0x02 0x03 0x04
0x203	1	1	0x1D 0x1E 0x1F 0x20
0x201	1	4	0x15 0x16 0x17 0x18
0x202	1	2	0x19 0x1A 0x1B 0x1C

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Write to C[0] = 0x3000
  - Miss, so retrieve from memory
- Problem: How to keep the cache and main memory consistent when you write data?

Tag	Valid	LRU	Data
0x100	1	3	0x01 0x02 0x03 0x04
0x203	1	1	0x1D 0x1E 0x1F 0x20
0x201	1	4	0x15 0x16 0x17 0x18
0x202	1	2	0x19 0x1A 0x1B 0x1C

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Write Policy

- **Write-through**
  - When a write occurs, update the data both in the cache and in main memory
  - Slows down writes a lot, since we need to write to main memory every time
    - Works better with multiple levels of caches (Thursday)
- **Write-back**
  - When a write occurs, only update the data in the cache
  - When we later evict the block, write all changes to main memory at the same time
  - Needs a "dirty" bit to signify that the block has to be written to memory
  - Need to be careful that we don't access main memory before we write back our data, especially if we're multithreading
  - Much faster than write-through
- Let's rerun the simulation with a write-back cache, and after we transpose B.

Caching Example: Write-Back Fully Associative with LRU

- Start: Cache starts cold
  - Note: New Dirty field
  - Note: Matrix B has been transposed into Bt to optimize cache efficiency
- 0 misses, 0 hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	0	0	0	0xBF 0x16 0x88 0x2B
0x733	0	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

A

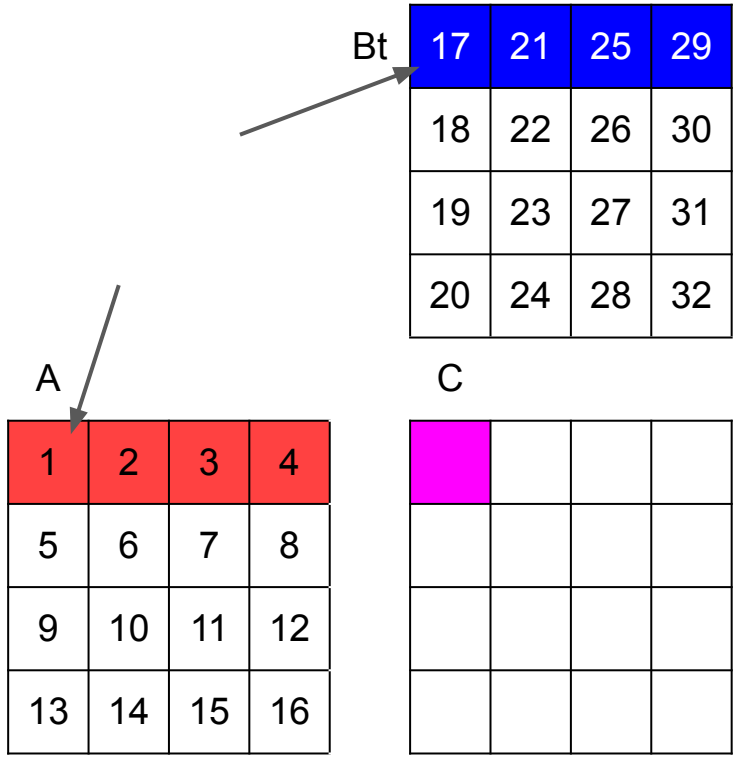
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Access A[0] = 0x1000
  - Miss
- Access Bt[0] = 0x2000
  - Miss

Tag	V	Dirty	LRU	Data
0x100	**0**	**0**	**0**	**0xBF 0x16 0x88 0x2B**
0x733	**0**	**0**	**0**	**0x3B 0x18 0xF1 0xB3**
0x156	**0**	**0**	**0**	**0xE6 0x57 0x49 0xEE**
0x4E9	**0**	**0**	**0**	**0xB5 0x81 0x67 0x3F**



Caching Example: Write-Back Fully Associative with LRU

- A[1]: Hit Bt[1]: Hit
- A[2]: Hit Bt[2]: Hit
- A[3]: Hit Bt[3]: Hit
- 6 Hits, 2 Misses!

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	2	0x01 0x02 0x03 0x04
0x200	1	0	1	0x11 0x15 0x19 0x1D
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Write to C[0] = 0x3000
  - Miss + Write, so set dirty bit
- Main memory not updated yet

Tag	V	Dirty	LRU	Data
0x100	1	0	2	0x01 0x02 0x03 0x04
0x200	1	0	1	0x11 0x15 0x19 0x1D
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

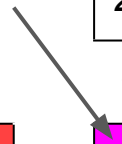
A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

C



Caching Example: Write-Back Fully Associative with LRU

- A[0]: Hit, Bt[4]: Miss
- …
- A[3]: Hit, Bt[7]: Hit.
- Write C[1]: Hit

- 8 Hits, 1 Miss here
- Total: 4 Misses, 14 Hits

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x200	1	0	2	0x11 0x15 0x19 0x1D
0x300	1	1	1	0xFA 0x?? 0x?? 0x??
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[0]: Hit, Bt[8]: Miss, evict block 0x200,
- …
- Write C[2]: Hit
- 8 Hits, 1 Miss here
- Totals: 5 Misses, 22 Hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x200	1	0	4	0x11 0x15 0x19 0x1D
0x300	1	1	1	0xFA 0x04 0x?? 0x??
0x201	1	0	2	0x12 0x16 0x1A 0x1E

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[0]-A[3], B[16]-B[19], C[3]:
  - 8 hits, 1 miss, block 0x201 evicted
- 8 Hits, 1 Miss here
- Totals: 6 Misses, 30 Hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x202	1	0	2	0x13 0x17 0x1B 0x1F
0x300	1	1	1	0xFA 0x04 0x0E 0x??
0x201	1	0	4	0x12 0x16 0x1A 0x1E

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[4]: Miss, evict block 0x202
- Bt[0]: Miss, evict block 0x100
- A[5]-A[7], Bt[1]-Bt[3]: 6 hits
- Total: 8 misses, 36 hits

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x300	1	1	1	0xFA 0x04 0x0E 0x18
0x203	1	0	2	0x14 0x18 0x1C 0x20

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Write C[4]: Miss, Write to dirty block, evict block 0x203
- A[4]: Hit
- Total: 9 misses, 37 hits

Tag	V	Dirty	LRU	Data
0x200	1	0	1	0x11 0x15 0x19 0x1D
0x101	1	0	2	0x05 0x06 0x07 0x08
0x300	1	1	3	0xFA 0x04 0x0E 0x18
0x203	1	0	4	0x14 0x18 0x1C 0x20

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Bt[4]: Miss, evict 0x300
  - Evicted a dirty block: only at this point does main memory get updated
- Total: 10 misses, 37 hits, 1 write-back

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x200	1	0	3	0x11 0x15 0x19 0x1D
0x101	1	0	1	0x05 0x06 0x07 0x08
0x300	1	1	4	0xFA 0x04 0x0E 0x18
0x301	1	1	2	0x6A 0x?? 0x?? 0x??

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[5]-Write C[5]: 7 hits
- Compute C[6]: 1 miss, 8 hits
- Compute C[7]: 1 miss, 8 hits
- Total: 12 misses, 60 hits, 1 write-back

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x200	1	0	4	0x11 0x15 0x19 0x1D
0x101	1	0	2	0x05 0x06 0x07 0x08
0x201	1	0	1	0x12 0x16 0x1A 0x1E
0x301	1	1	3	0x6A 0x84 0x9E 0xB8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280

Caching Example: Write-Back Fully Associative with LRU

- Compute C[8]: 3 misses, 6 hits
- Compute C[9]: 1 miss, 8 hits, 1 write-back
- Compute C[10]: 1 miss, 8 hits
- Compute C[11]: 1 miss, 8 hits
- Total: 18 misses, 90 hits, 2 write-backs

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x101	1	0	3	0x05 0x06 0x07 0x08
0x203	1	0	2	0x14 0x18 0x1C 0x20
0x301	1	1	1	0x6A 0x84 0x9E 0xB8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280

Caching Example: Write-Back Fully Associative with LRU

- Compute C[12]: 3 misses, 6 hits
- Compute C[13]: 1 miss, 8 hits, 1 write-back
- Compute C[14]: 1 miss, 8 hits
- Compute C[15]: 1 miss, 8 hits
- Total: 24 misses, 120 hits, 3 write-backs

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x102	1	0	3	0x09 0x0A 0x0B 0x0C
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x302	1	1	1	0xDA 0x04 0x2E 0x58
0x203	1	0	2	0x14 0x18 0x1C 0x20

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618	658	670	696

Caching Example: Write-Back Fully Associative with LRU

- Final step: Flush the cache
  - Write back all dirty blocks
  - Invalidate all blocks
- Final Count: 24 misses, 120 hits, 4 write-backs

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x103	1	0	3	0x0D 0x0E 0x0F 0x10
0x203	1	0	2	0x14 0x18 0x1C 0x20
0x303	1	1	1	0x4A 0x84 0xBE 0xF8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618	658	670	696
986	1028	1070	1112

Caching Example: Write-Back Fully Associative with LRU

- Final Count: 24 misses, 120 hits, 4 write-backs
- 83% hit rate
  - Way better than the 50% we got before

Tag	V	Dirty	LRU	Data
0x202	0	0	4	0x13 0x17 0x1B 0x1F
0x103	0	0	3	0x0D 0x0E 0x0F 0x10
0x203	0	0	2	0x14 0x18 0x1C 0x20
0x303	0	0	1	0x4A 0x84 0xBE 0xF8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

C

250	260	270	280
618	658	670	696
986	1028	1070	1112
1354	1412	1470	1528

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
- Tomorrow: Different types of caches that sacrifice hit rate for hit time and hit speed