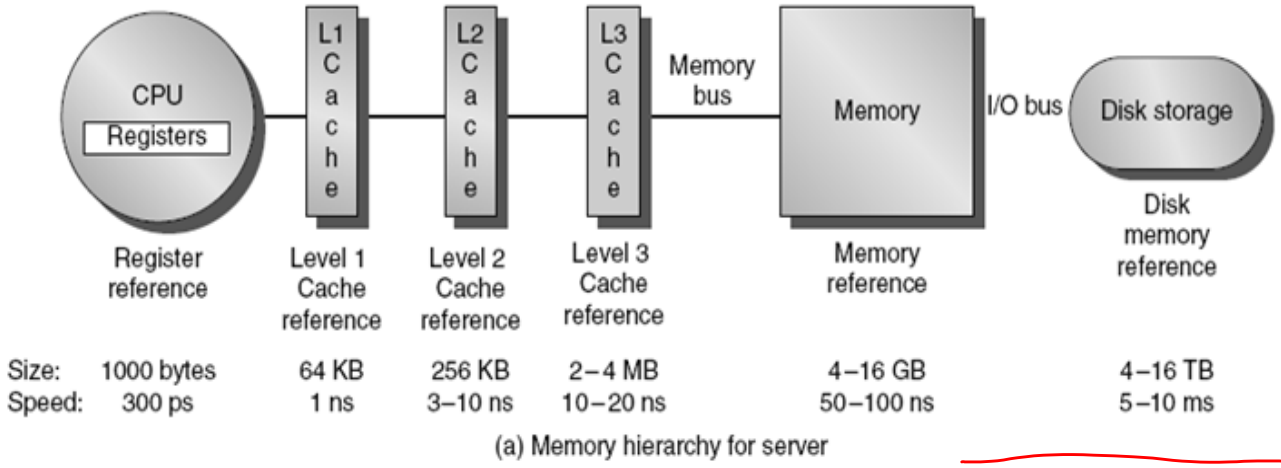
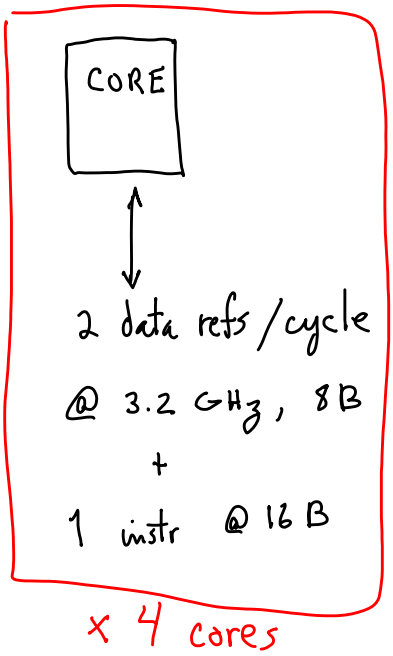
- Programmers want unlimited amounts of memory with low latency
- Fast memory technology is more expensive per bit than slower memory
- Solution: organize memory system into a hierarchy
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - Gives the allusion of a large, fast memory being presented to the processor



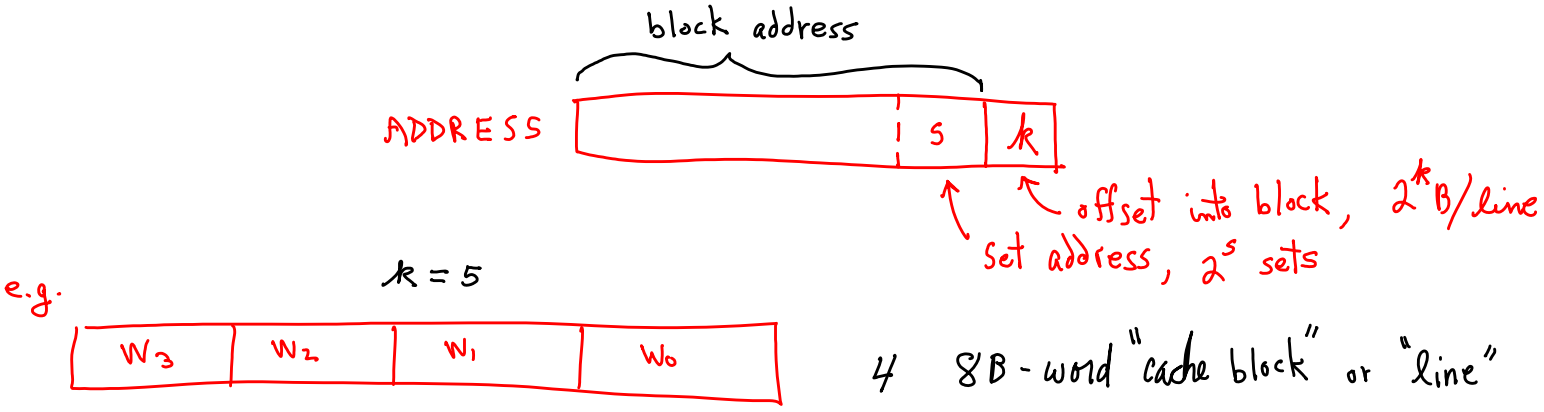
(a) Memory hierarchy for server

- Aggregate peak bandwidth grows with # cores:
  - Intel Core i7 can generate two references per core per clock
  - Four cores and 3.2 GHz clock
    - 25.6 billion 64-bit data references/second +
    - 12.8 billion 128-bit instruction references
    - = 409.6 GB/s!
  - DRAM bandwidth is only 6% of this (25 GB/s)
  - Requires:
    - Multi-port, pipelined caches
    - Two levels of cache per core
    - Shared third-level cache on chip



Metrics: Avg Access time = (hit rate)(hit time) + (miss rate)(miss time)
Avg Power = #(active devices)(avg dynamic power)

- When a word is not found in the cache, a *miss* occurs:
  - Fetch word from lower level in hierarchy, requiring a higher latency reference
  - Lower level may be another cache or the main memory
  - Also fetch the other words contained within the *block*
    - Takes advantage of spatial locality
  - Place block into cache in any location within its *set*, determined by address
    - block address MOD number of sets



- Miss rate
  - Fraction of cache access that result in a miss

- Causes of misses
  - Compulsory
    - First reference to a block
  - Capacity
    - Blocks discarded and later retrieved
  - Conflict
    - Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache

+ Coherency + context switching

  - Note that speculative and multithreaded processors may execute other instructions during a miss
    - Reduces performance impact of misses

⇒ Cache handles miss while accessing other data/instructions

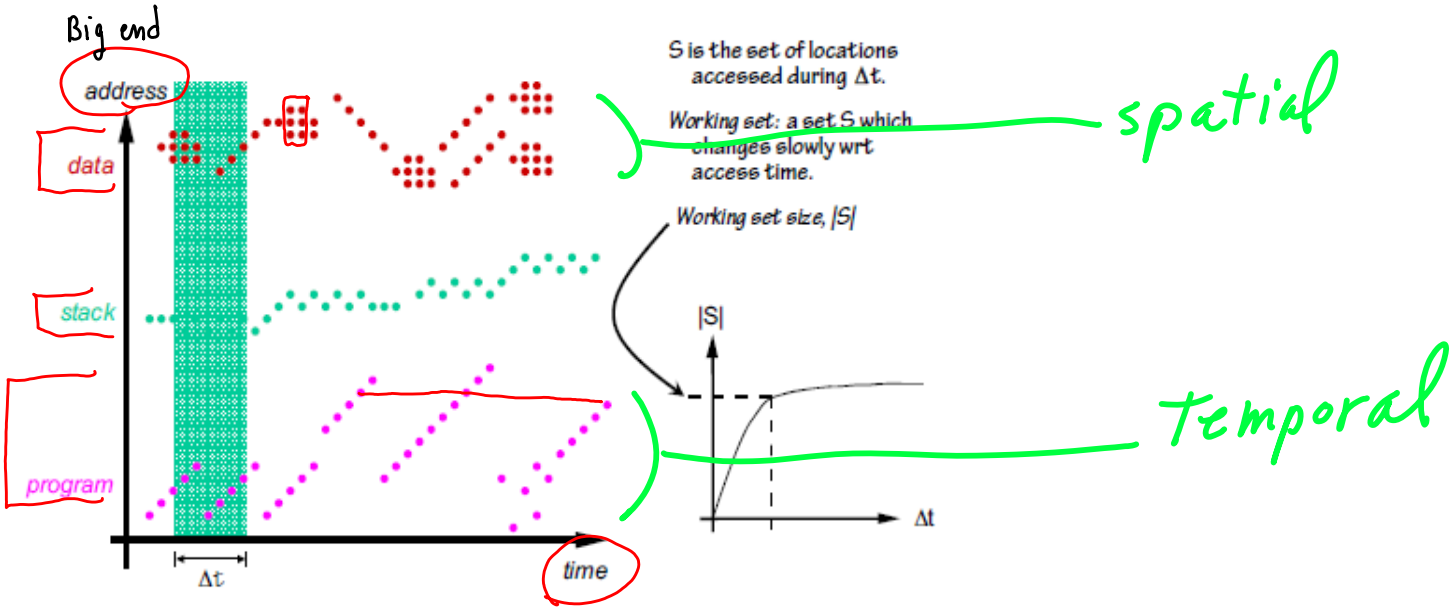
We can hide latency if we can

1. Do more work than we do Reads and Writes

2. Re-use things by keeping them handy

Locality

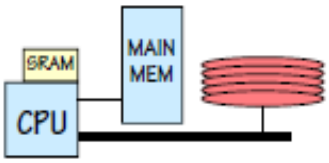
Memory Reference Patterns



Exploiting the Memory Hierarchy

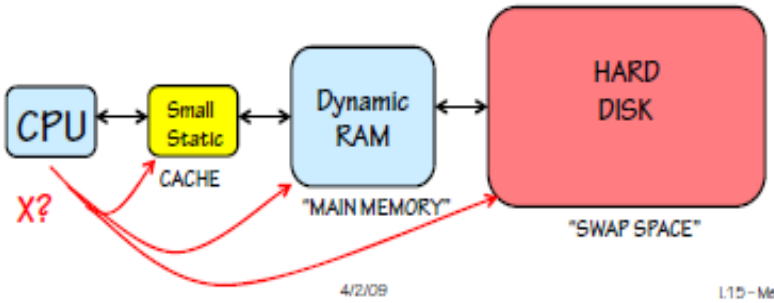
Approach 1 (Cray, others): *Expose* Hierarchy

- Registers, Main Memory, Disk each available as storage alternatives;
- Tell programmers: "Use them cleverly"

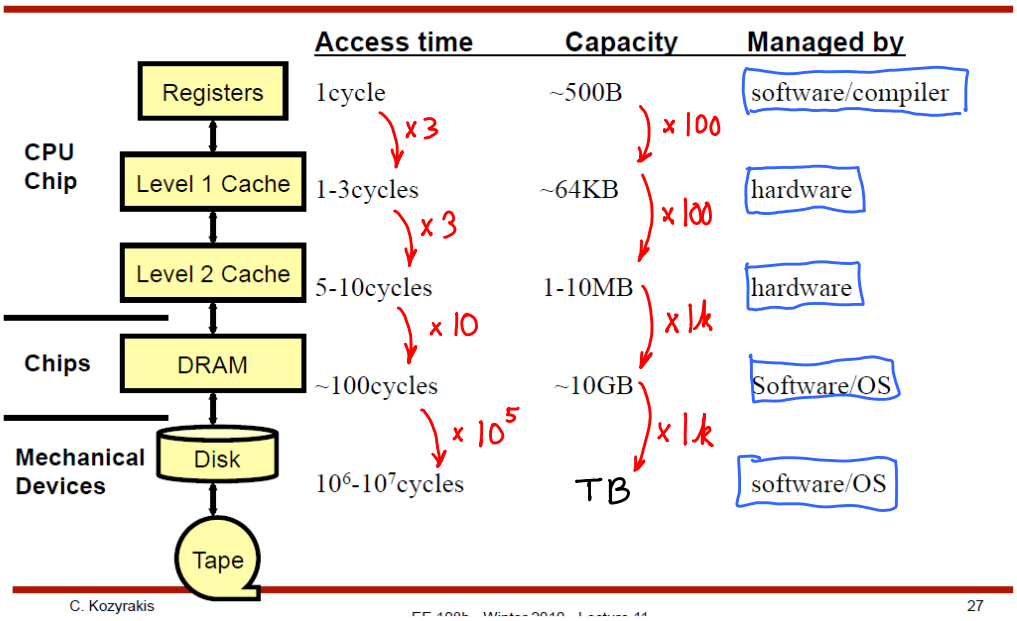


Approach 2: *Hide* Hierarchy

- Programming model: SINGLE kind of memory, single address space.
- Machine AUTOMATICALLY assigns locations to fast or slow memory, depending on usage patterns.



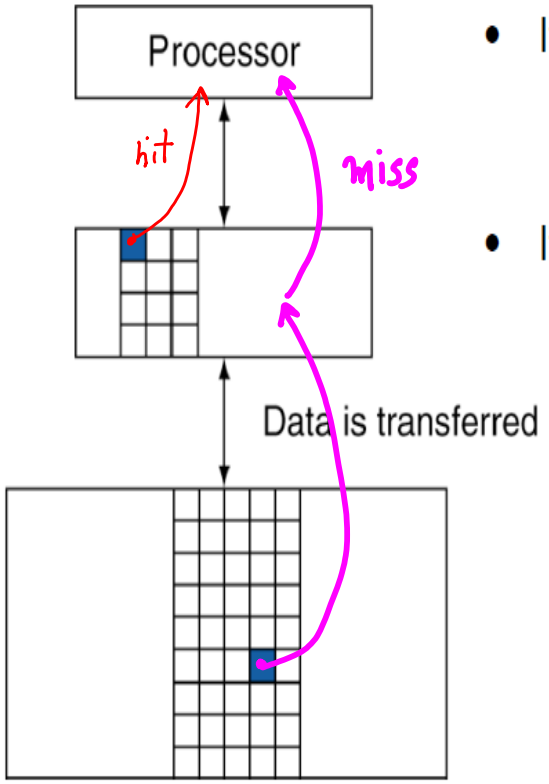
Typical Memory Hierarchy: Everything is a Cache for Something Else



L3?
L4?

Larger gap
→ Therefore? do what?

Hide latency.
using what?



- 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 HR
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio

$\frac{N_{misses}}{N_{accesses}} = MR$

$HR = N_{hits} / N_{accesses}$

  - Then data supplied to CPU from upper level

--- What size is appropriate?
--- Is flexibility important?

- Need to define an average access time
  - Since some will be fast and some slow

    Access time = hit time + miss rate x miss penalty

  - The hope is that the hit time will be low and the miss rate low since the miss penalty is so much larger than the hit time

% hits

time to R/W data + cache hit time

(hit rate)(hit time) + (miss rate)(miss time)

$= (1-MR)\,T_{hit} + MR(T_{access} + T_{hit})$

$= T_{hit}((1-MR) + MR) + MR\,T_{access}$

$= T_{hit} + MR(T_{access})$

miss Penalty

- Average Memory Access Time (AMAT)
  - Formula can be applied to any level of the hierarchy
    - Access time for that level
  - Can be generalized for the entire hierarchy
    - Average access time that the processor sees for a reference

what's important?
Overall access time, averaged over all levels and all instructions.

How Processor Handles a Miss

- Assume that cache access occurs in 1 cycle
  - Hit is great, and basic pipeline is fine

    CPI penalty = miss rate x miss penalty

$CPI_{penalty}\,(cycles) = MR \cdot T_{penalty}\,(sec)\,CR\left(\frac{cycles}{sec}\right)$

- A miss stalls the pipeline (for a instruction or data miss)
  - Stall the pipeline (you don't have the data it needs)
  - Send the address that missed to the memory
  - Instruct main memory to perform a read and wait
  - When access completes, return the data to the processor
  - Restart the instruction

COMPLETE

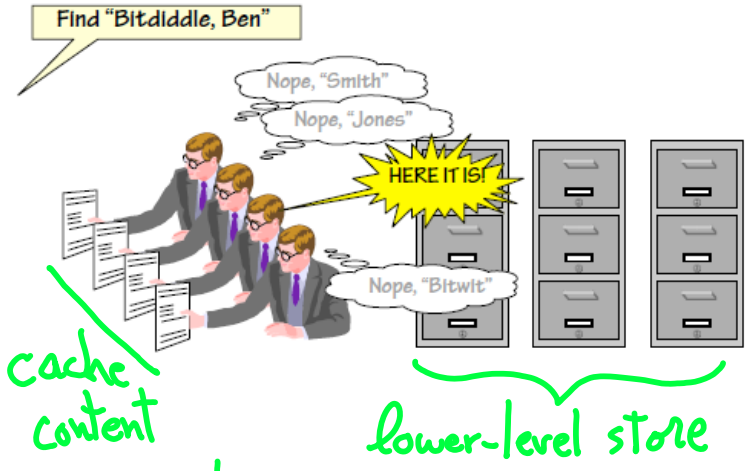
Generally
Turing Tape, move L/R, copy region, But w/ distance cost

How To Build A Cache?

- Big question: locating data in the cache
  - I need to map a large address space into a small memory
- How do I do that?
  - Can build full associative lookup in hardware, but complex
  - Need to find a simple but effective solution
  - Two common techniques:

    OR
    - Direct Mapped
    - Set Associative
- Further questions ?
  - Block size (crucial for spatial locality) how big?
  - Replacement policy (crucial for temporal locality) when/what?
  - Write policy (writes are always more complex) when/what?

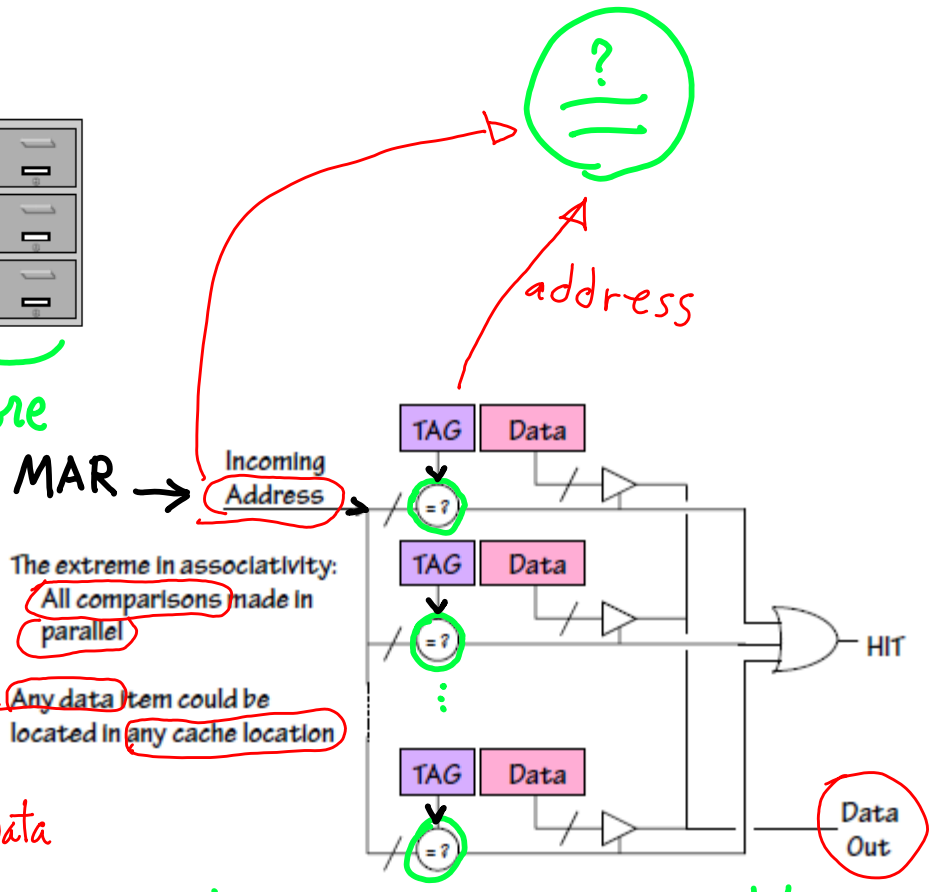
Associativity: Parallel Lookup



cache content

lower-level store

Look at all at same time



The extreme in associativity:
All comparisons made in parallel

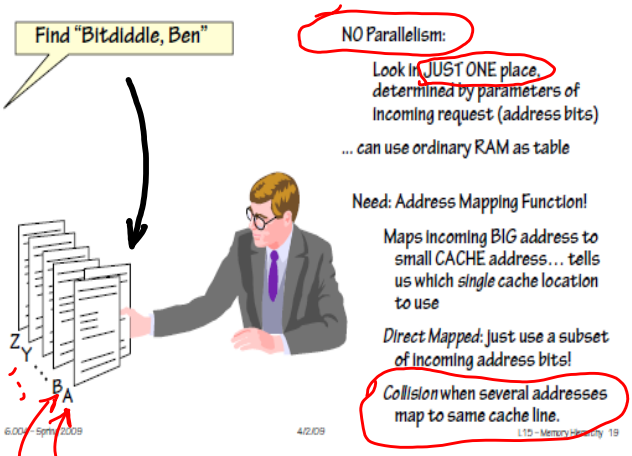
Any data item could be located in any cache location

item can be in any slot; load any empty.

MAR | Tag

Some portion of address stored w/ data

Direct-Mapped Cache
(non-associative)



A slot

B slot

Look at exactly one.

PROBLEM:

Contention among B's.... each competes for same cache line!

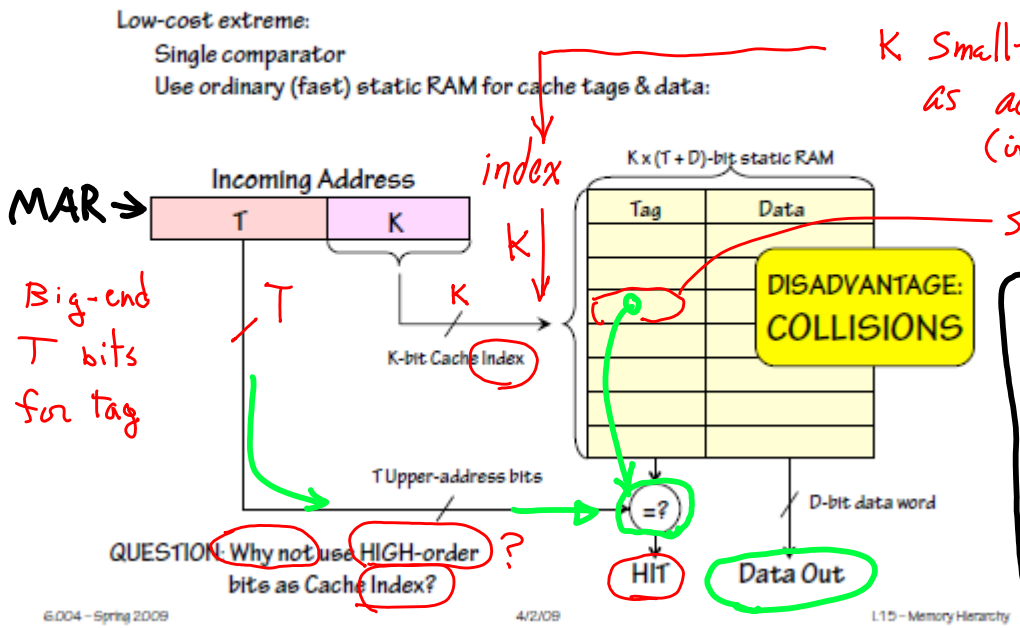
CAN'T cache both "Bitdiddle" & "Bitwit"

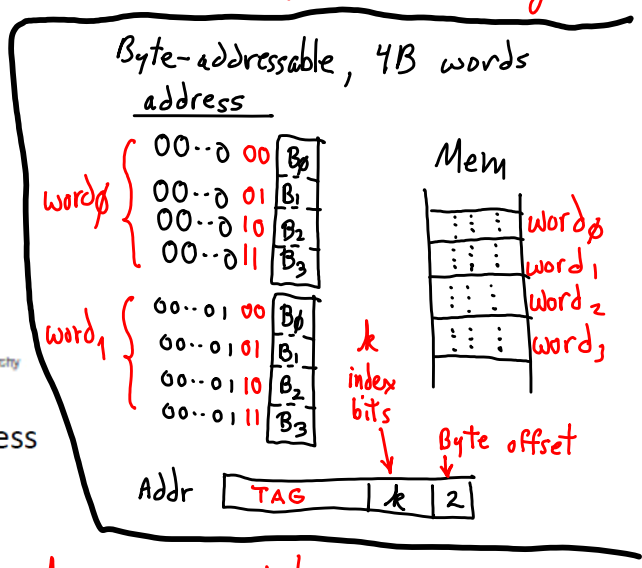
... Suppose B's tend to come at once?

contention 1 slot for B's

BETTER IDEA: File by LAST letter! ?

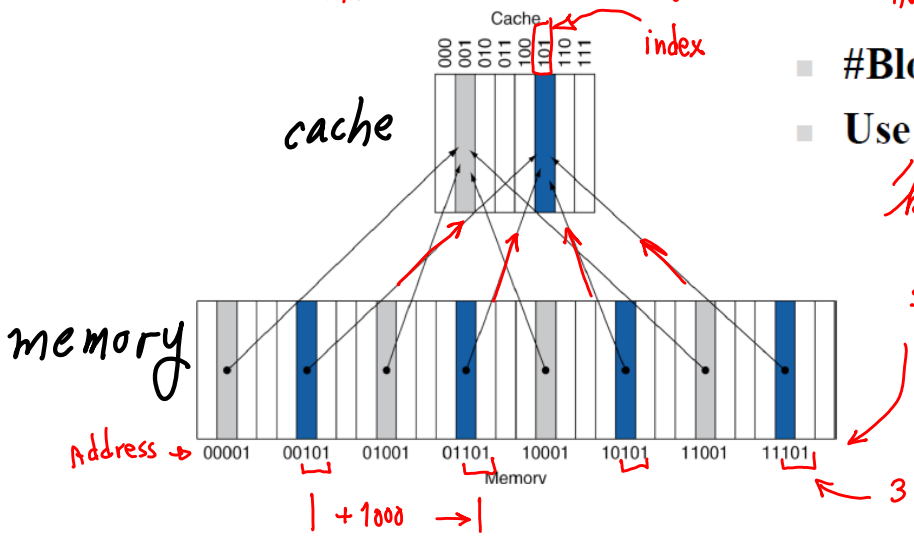
Direct Mapped Cache





- Location in cache determined by (main) memory address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

small-end 3-bits of address = index of cache slot



- **#Blocks is a power of 2**
- **Use low-order address bits**

k for k-bit index

spreads collisions by 001000 (k index bits)

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits (Biggend bits)
  - Called the tag

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

E.G.

- 8-blocks, 1 word/block, direct mapped
- Initial state

START up?
- boot, how?
- what's in SRAM?

Process swap?
- system actions?

Index	V	Tag	Data
000	N	?	?
001	N	?	?
010	N	?	?
011	N	?	?
100	N	?	?
101	N	?	?
110	N	?	?
111	N	?	?

3-bits for index (ignore bits for byte offset into word)

LW $1, 0($2)

10110 $2 R2

MAR

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Compulsory/Cold Miss

index

Index	V	Tag	Data /instr
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110] = 1234
111	N		

write Tag

write Data from mem

mixed cache?
data cache?
instruction cache?

Can we avoid cold misses?

!VALID → Miss
mem fetch, stall, load, restart

LW $3, 0($4)

11010 $4 R4

MAR

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Compulsory/Cold Miss

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010] = ABCD
011	N		
100	N		
101	N		
110	Y	10	Mem[10110] = 1234
111	N		

data

!VALID
fetch stall load restart

MAR

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Hit

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

write To Mem

write To cache

5678 ABCD

SW $5, 0($4)

5678 $5/R5

11010 $4 R4

valid + Tag-match = hit

MAR

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Replacement

write new tag

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

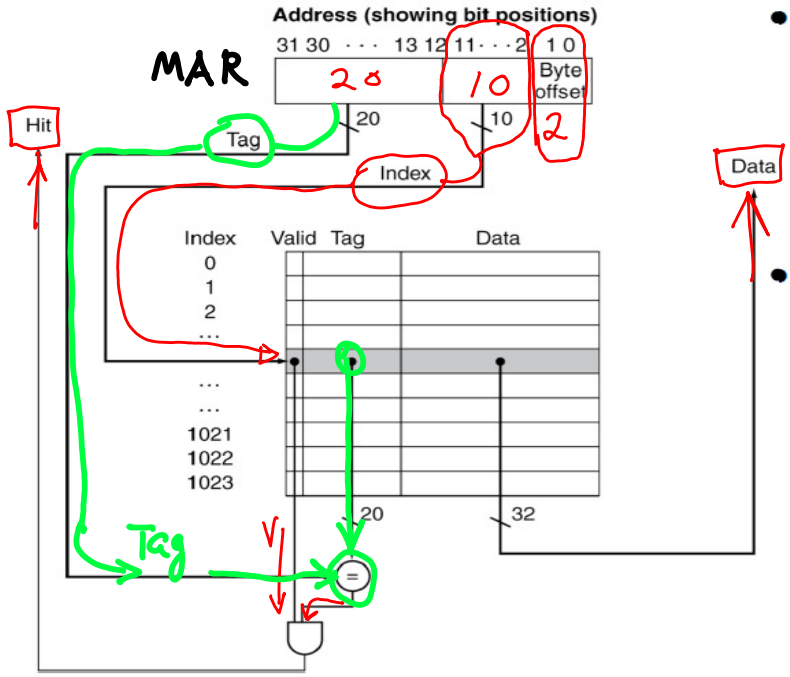
collision 12BF 5678

SW $7, 0($8)

12BF $7 10010 $8

valid + Tag-mismatch = miss, collision

Write prior data to memory? Read data from memory, then write cache?



Byte offset ignored

- Assumptions
  - 32-bit address
  - 4 Kbyte cache → 2^{10} blocks, @ 2^2 B
  - 1024 blocks, 1 word/block
- Steps
  1. Use index to read V, tag from cache
  2. Compare read tag with tag from address
  3. If match, return data & hit signal
  4. Otherwise, return miss

Direct Mapped Problems: Conflict misses

- Two blocks that are used concurrently and map to same index
  - Only one can fit in the cache, regardless of cache size
  - No flexibility in placing 2nd block elsewhere

- ~~Trashing~~ Thrashing
  - If accesses alternate, one block will replace the other before reuse
  - No benefit from caching worse than no cache

Same Tags

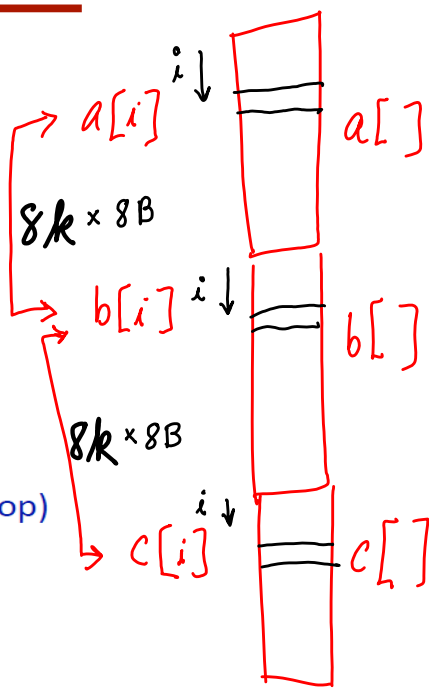
LW (1100110)
LW (0101110)
SW (1100110)
SW (0101110)

- Consider the following example code:

1k x 8B cache
= 2^{13} B = 8 kB

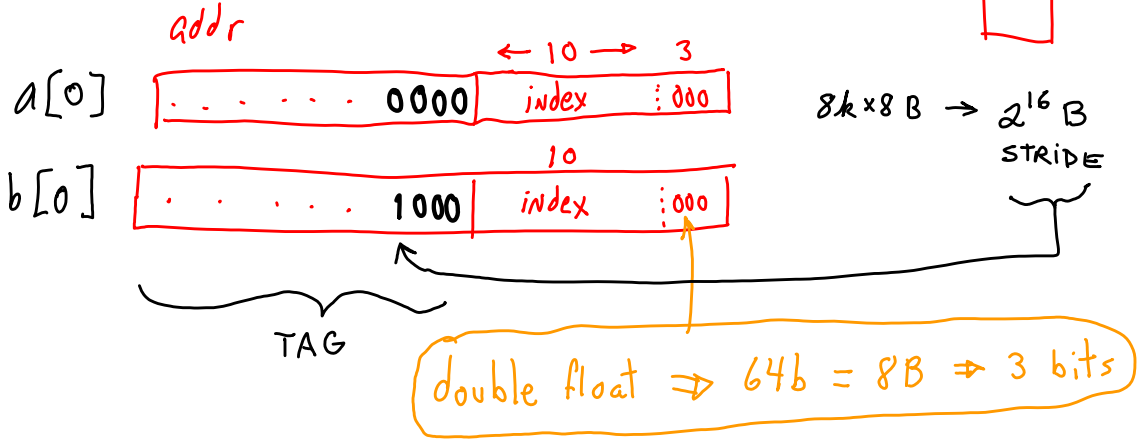
```
double a[8192], b[8192], c[8192];
        8k x 64b   8k      8k
void vector_sum()
{
    int i;

    for (i = 0; i < 8192; i++)
        c[i] = a[i] + b[i];
}
```



  - Arrays a, b, and c will tend to conflict in small caches
  - Code will get cache misses with *every* array access (3 per loop)
  - Spatial locality savings from blocks will be eliminated
- How can the severity of the conflicts be reduced?

Any stride in increments of 2^10 X 2^3 Bytes causes same problem: indices are identical, but tags differ.



How can we fix this?

index bits > 10?
How big a cache?

index bits: 10 13 15
size: 1k x 8B ... 8k x 8B ... 4 x 8k x 8B
all collide all collide no collisions

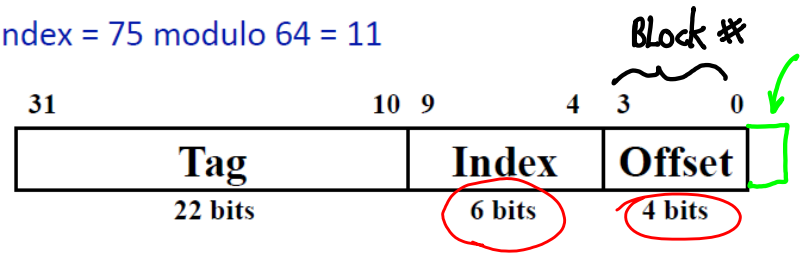
Programmer's mistake?

How to make system crawl, worst case? } Let's have a contest!

Larger Block Size → n words / Block

- Motivation: exploit spatial locality & amortize overheads
- This example: 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
  - Block address = 1200/16 = 75
  - Block index = 75 modulo 64 = 11

(1 latency) / (n words)



8k B cache

64 Blocks 16 64b words

$(2^6 \text{ Blocks})(2^4)(2^3 \text{ B})$

- What is the impact of larger blocks on tag/index size?
- What is the impact of larger blocks on the cache overhead?
  - Overhead = tags & valid bits

extra bits stored per block: tag, V

$(22 + 1)\ 2^6 \Rightarrow 23 \cdot 2^6 \text{ b} \sim 200\text{B}$ ~2.5%

vs

$(22 + 1)\ 2^{10} \Rightarrow 23 \cdot 2^{10} \text{ b} \sim 3\text{kB}$ ~38%

* cache lines

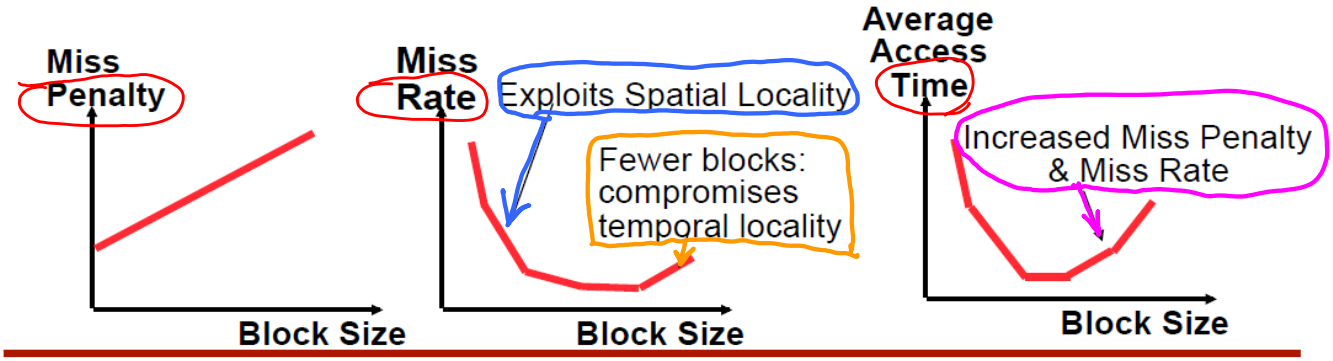
vs

1k Blocks 1 64b Word

$(2^{10} \text{ Blocks})(2^0 \text{ B})(2^3 \text{ B})$

- Larger block sizes take advantage of spatial locality
  - Also incurs larger miss penalty since it takes longer to transfer the block into the cache
  - Large block can also increase the average time or the miss rate
- Tradeoff in selecting block size
- Average Access Time = Hit Time • (1-MR) + Miss Penalty • MR

depends on organization + width



Assume fixed { Bandwidth to memory; Total cache size: # Blocks = Total size / Block size }

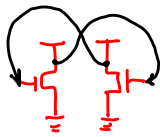
Fully-assoc. vs. Direct-mapped

Fully-associative N-line cache:

- N tag comparators, registers used for tag/data storage ($$$)
- Location A might be cached in any one of the N cache lines; no restrictions!
- Replacement strategy (e.g., LRU) used to pick which line to use when loading new word(s) into cache
- PROBLEM: Cost!

implement policy

Direct-mapped N-line cache:



- 1 tag comparator, SRAM used for tag/data storage ($)
- Location A is cached in a specific line of the cache determined by its address; address "collisions" possible
- Replacement strategy not needed: each word can only be cached in one specific cache line
- PROBLEM: Contention!

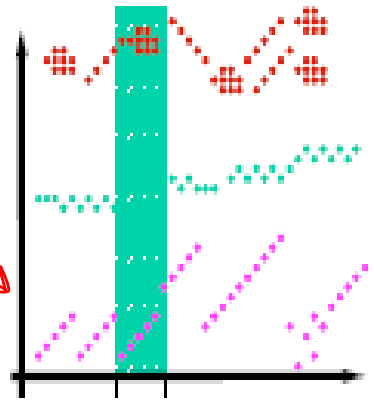
cheap!
Fast data access on hits

Cost vs Contention

two observations...

1. Probability of collision diminishes with cache size...
   ... so lets build HUGE direct-mapped caches, using cheap SRAM!
2. Contention mostly occurs between independent "hot spots" --
   - Instruction fetches vs stack frame vs data structures, etc
   - Ability to simultaneously cache a few (2? 4? 8?) hot spots eliminates most collisions

   ... so lets build caches that allow each location to be stored in some restricted set of cache lines, rather than in exactly one (direct mapped) or every line (fully associative).



Insight: an N-way set-associative cache affords modest parallelism

- parallel lookup (associativity): restricted to small set of N lines
- modest parallelism deals with most contention at modest cost
- can implement using N direct-mapped caches, running in parallel

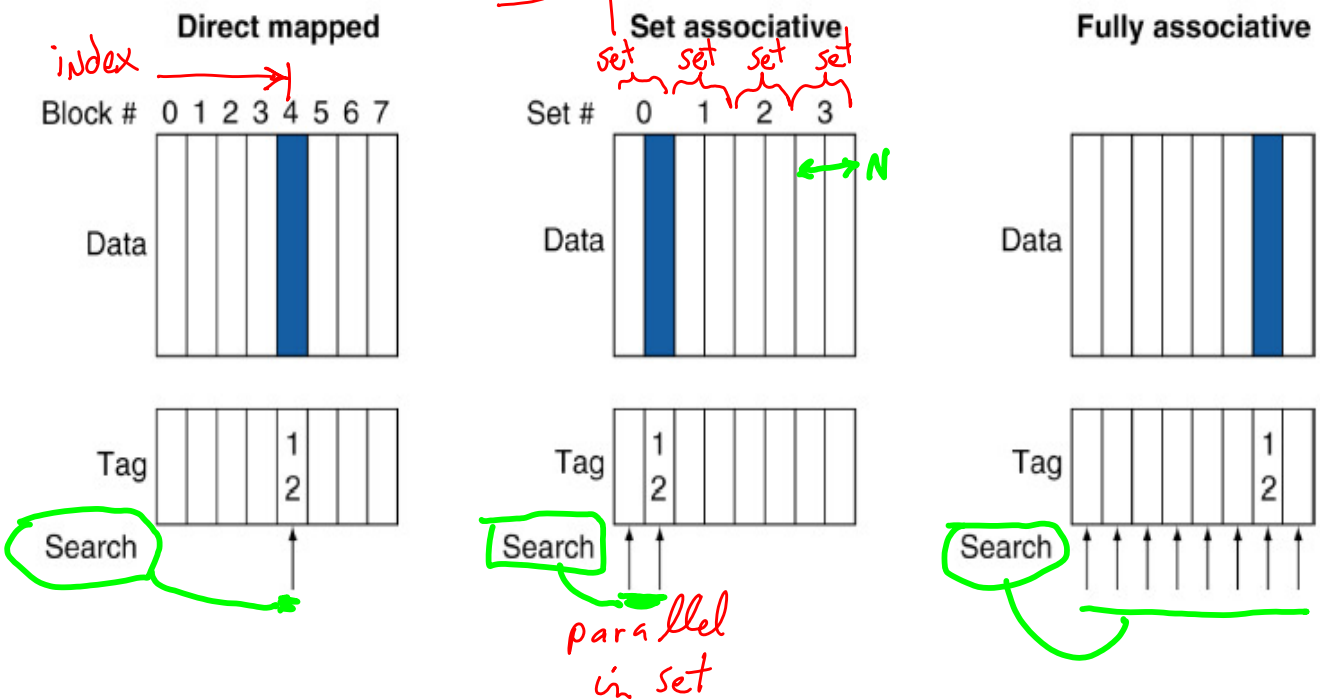
N-way Set Associative

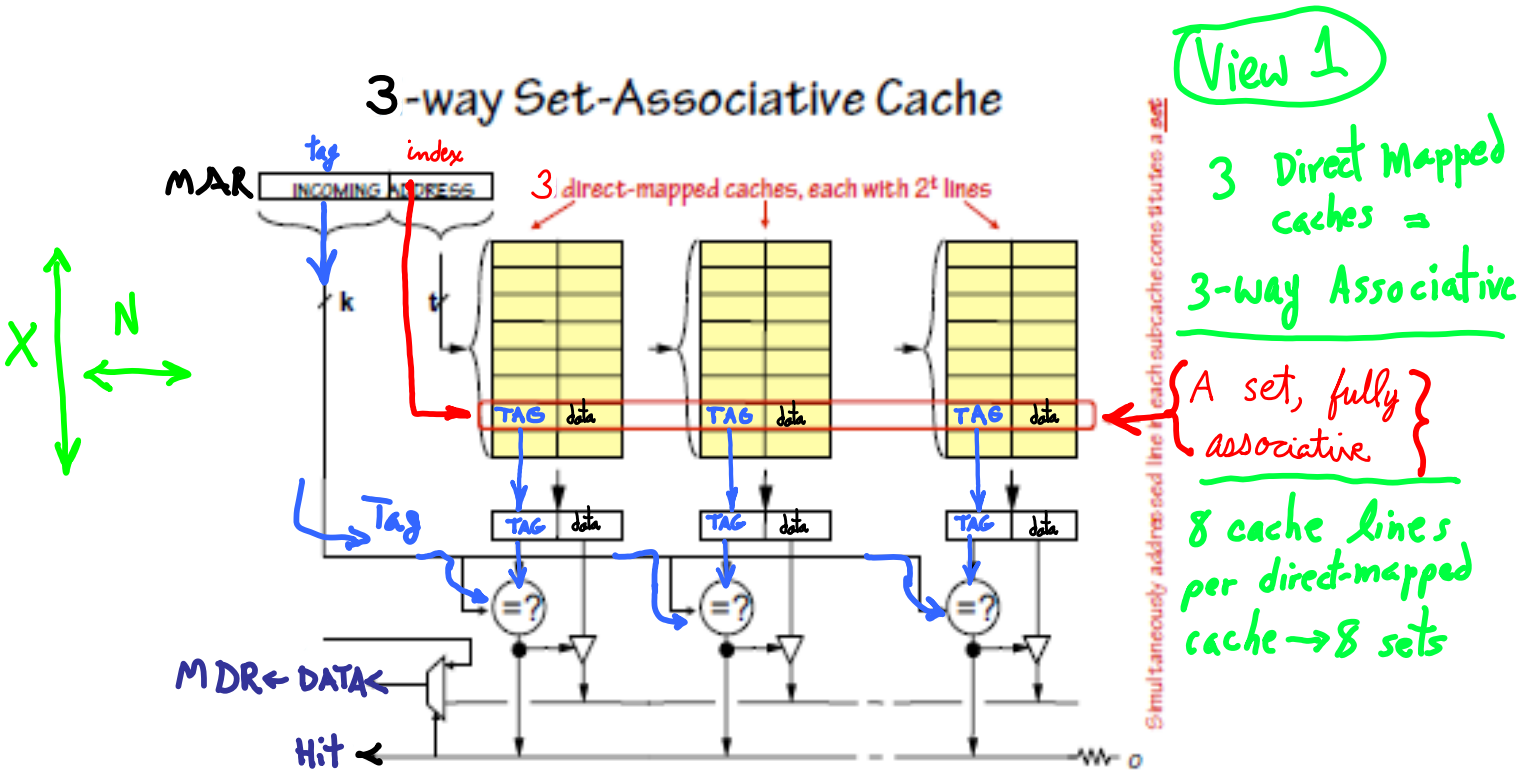
- Compromise between direct-mapped and fully associative
  - Each memory block can go to one of N entries in cache
    - Each "set" can store N blocks; a cache contains some number of sets
  - For fast access, all blocks in a set are search in parallel

- How to think of a N-way associative cache with X sets
  - 1st view: N direct mapped caches each with X entries
    - Caches search in parallel
    - Need to coordinate on data output and signaling hit/miss
  - 2nd view: X fully associative caches each with N entries each
    - One cache searched in each case

View 2

X = 4 Fully Associative (2-way)





E. G.

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative (vs. vs.)
  - Block access sequence: 0, 8, 0, 6, 8 — 3 different block addresses

- Direct mapped

index

ADDRESS	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
000	0	miss	Mem[0]			
800	0	miss	Mem[8]			
000	0	miss	Mem[0]			
610	2	miss	Mem[0]		Mem[6]	
800	0	miss	Mem[8]		Mem[6]	

4-block cache at t = 0

Collision

collision

Collision

Time

2-bit index

tag

- 2-way set associative

ADDRESS	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0 0 0	0	miss	Mem[0]			
8 0 0	0	miss	Mem[0]	Mem[8]		
0 0 0	0	hit	Mem[0]	Mem[8]		
6 1 0	0	miss	Mem[0]	Mem[6]		
8 0 0	0	miss	Mem[8]	Mem[6]		

TIME

LRU

Capacity

tag

1-bit index

- Fully associative

NO index

ADDRESS		Hit/miss	Cache content after access			
0 00		miss	Mem[0]			
8 00		miss	Mem[0]	Mem[8]		
0 00		hit	Mem[0]	Mem[8]		
6 10		miss	Mem[0]	Mem[8]	Mem[6]	
8 00		hit	Mem[0]	Mem[8]	Mem[6]	

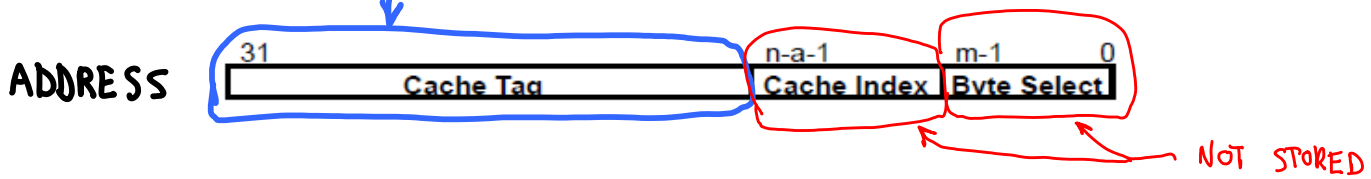
Time

Tag

0-bit index

SPACE OVERHEAD ?

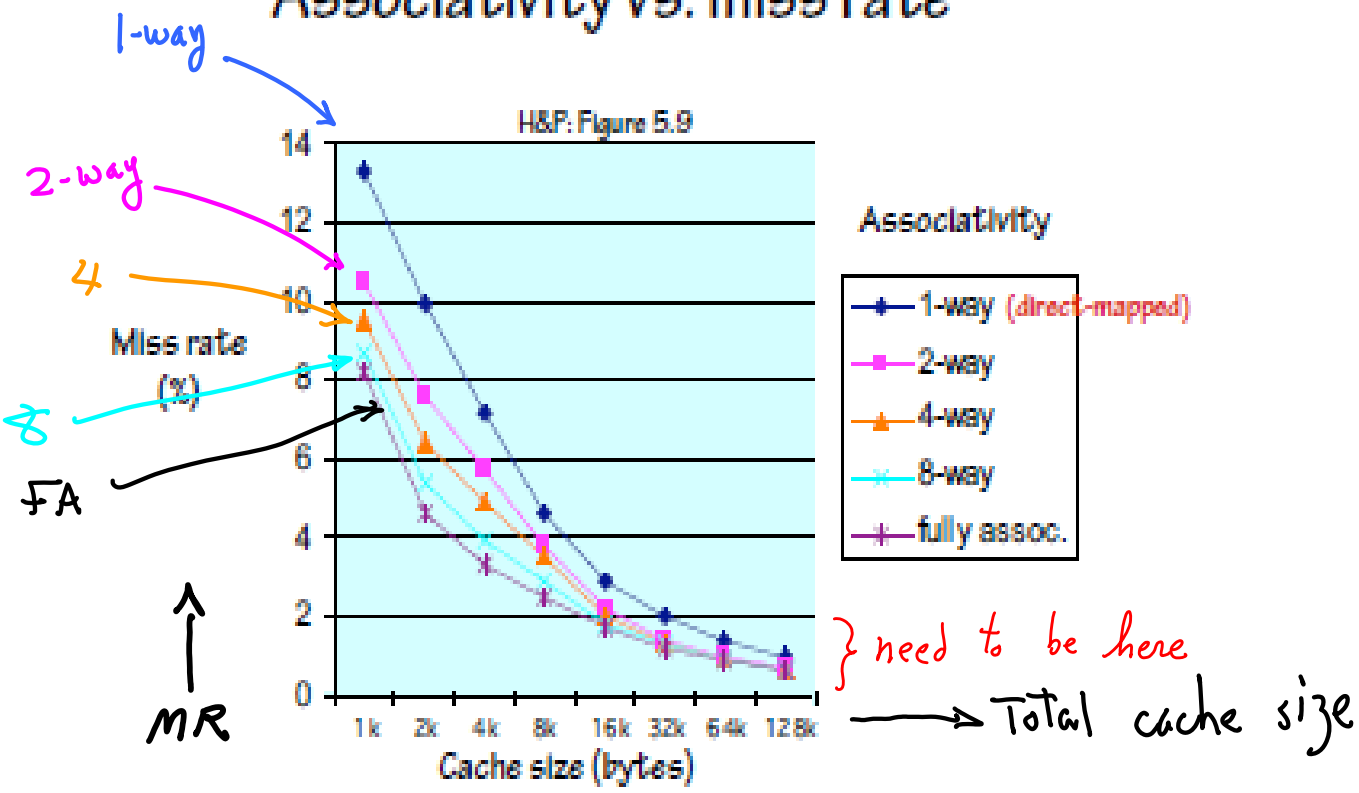
STORED IN CACHE



- Observation
  - For fixed size, length of tags increases with the associativity
  - Associative caches incur more overhead for tags

N

Associativity vs. miss rate



- 8-way is (almost) as effective as fully-associative
- rule of thumb: N-line direct-mapped == N/2-line 2-way set assoc.

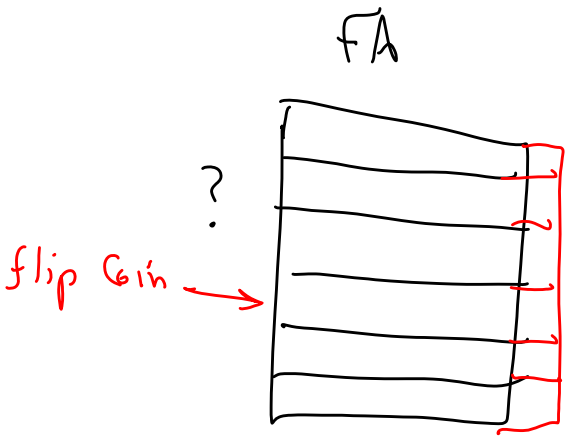
Another Job mix

- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

↓ diminishing returns ?

Replacement Methods

- Which line do you replace on a miss?
- Direct Mapped
  - Easy, you have only one choice
  - Replace the line at the index you need
- N-way Set Associative
  - Need to choose which way to replace
  - Random (choose one at random)
  - Least Recently Used (LRU) (the one used least recently)
    - Often difficult to calculate, so people use approximations. Often they are really not recently used



Handling of WRITES

workload { How many reads? How many writes? How many read-after-write?

Observation: Most (90+%) of memory accesses are *READs*. How should we handle writes? Issues:

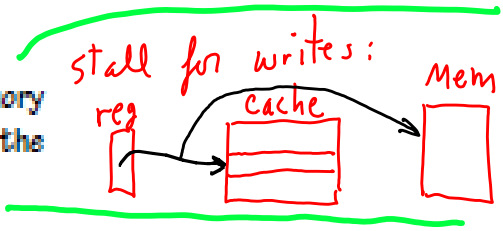
Write-through: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds "the truth".

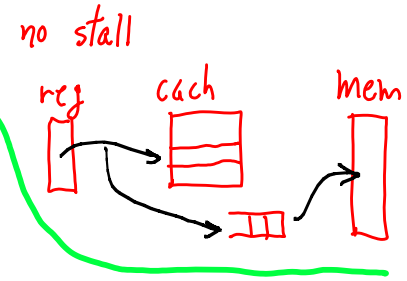
Write-behind: CPU writes are cached; writes to main memory may be buffered, perhaps pipelined. CPU keeps executing while writes are completed (in order) in the background.

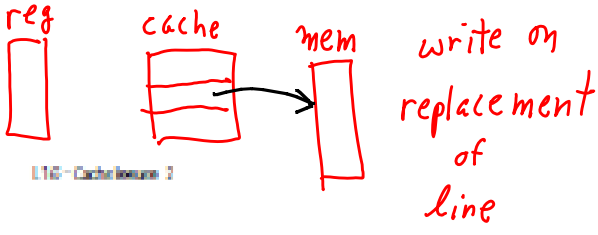
Write-back: CPU writes are cached, but not immediately written to main memory. Memory contents can be "stale".

Our cache thus far uses write-through.

Can we improve write performance?







- Interesting observation
  - Processor does not need to "wait" until the store completes

- Write-through (write data go to cache and memory)
  - Main memory is updated on each cache write
  - Replacing a cache entry is simple (just overwrite new block)
  - Memory write causes significant delay if pipeline must stall

memory updated → consistent
writes to cache clobber data in cache (prior writes already in mem)
– STALLS
– HIGH MEM USAGE

- Write-back (write data only goes to the cache)
  - Only the cache entry is updated on each cache write so main memory and the cache data are inconsistent
  - Add "dirty" bit to the cache entry to indicate whether the data in the cache entry must be committed to memory
  - Replacing a cache entry requires writing the data back to memory before replacing the entry if it is "dirty"

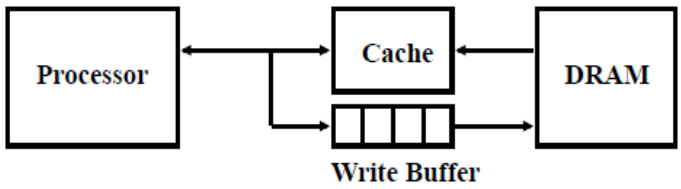
inconsistent memory
dirty-bit for replacement writes
+ fewer STALLS
+ LOW MEM USAGE

- Write-through
  - Misses are simpler and cheaper (no write-back to memory)
  - Easier to implement
    - Keep in mind for lab 4
    - But requires buffering to be practical (see following slide)
  - Uses a lot of bandwidth to the next level of memory

Use a write buffer?
+ writes are words
⇒ Lower bandwidth? But overall higher?

- Write-back
  - Writes are fast on a hit
  - Multiple writes within a block require only one "writeback" later
  - Efficient block transfer on write back to memory at eviction

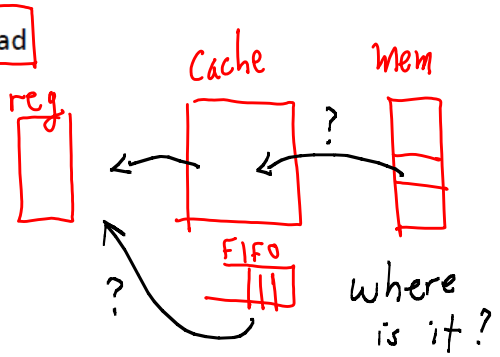
write backs are blocks
⇒ high mem bandwidth ☹
⇒ multiple writes ☺ per block
→ Low bandwidth



- Use Write Buffer between cache and memory
  - Processor writes data into the cache and the write buffer
  - Memory controller slowly "drains" buffer to memory

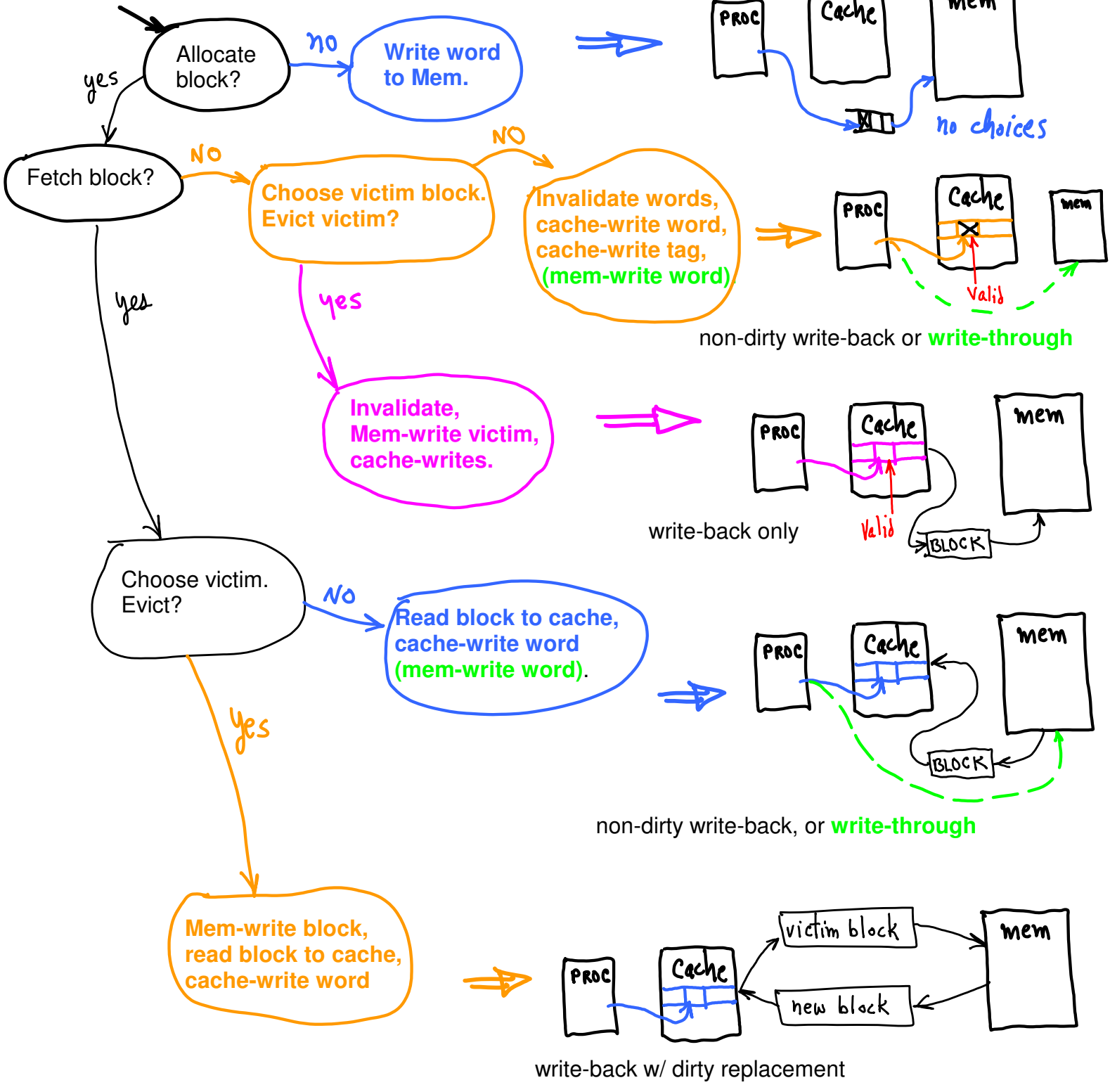
- Write Buffer: a first-in-first-out buffer (FIFO)
  - Typically holds a small number of writes
  - Can absorb small bursts as long as the long term rate of writing to the buffer does not exceed the maximum rate of writing to DRAM

- If a write-through cache has a write buffer, what should happen on a read miss?



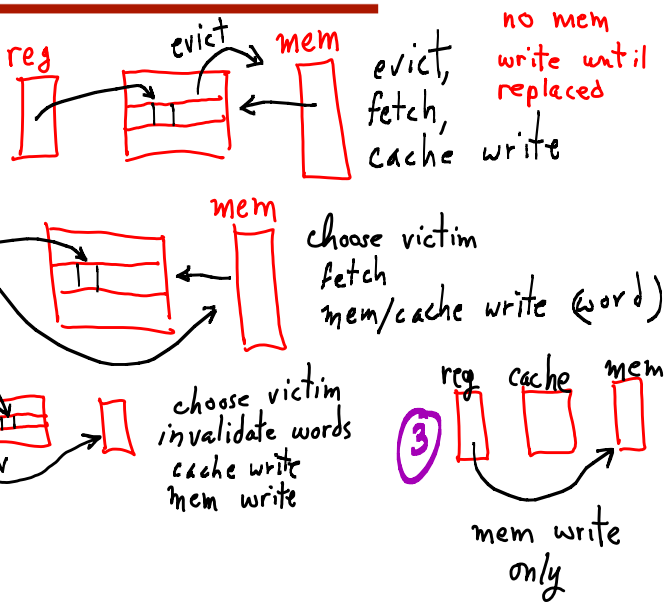
- Are write-buffers of any use for write-back caches?

what to do on write miss?



Write Miss — Typical Choices

- Write-back caches
  - Write-allocate, fetch-on-miss (why?)
- Write-through caches
  1. Write-allocate, fetch-on-miss
  2. Write-allocate, no-fetch-on-miss
  3. No-write-allocate, write-around



- Which program patterns match each policy?
- Modern HW support multiple policies
  - Selected by OS on at some coarse granularity (e.g. 4KB)

Be Careful, Even with Write Hits

- **Reading** from a cache
  - Read tags and data in parallel
  - If it hits, return the data, else go to lower level

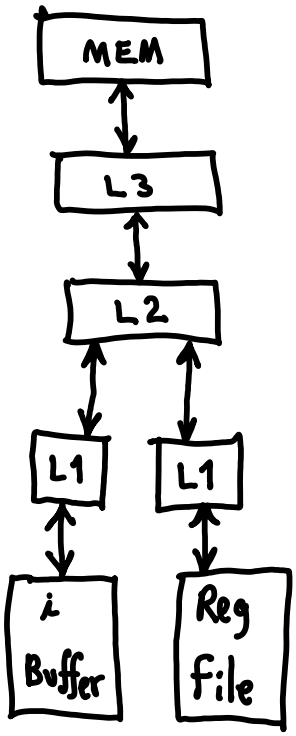
1. Read (Tag, data)
(Stall or No Stall)

- **Writing** a cache can take more time ?
  - First read tag to determine hit/miss (access 1)
  - Then overwrite data on a hit (access 2)
    - Otherwise, you may overwrite dirty data or write the wrong cache way

1. Read (Tag) (stall or no stall)
2. write data

- Can you ever access tag and write data in parallel?

Splitting Caches



- Most processors have separate caches for instructions & data
  - Often noted as $I and $D ☺

IMEM
DMEM

- Advantages
  - Extra access port
  - Can customize to specific access patterns
  - Low hit time
- Disadvantages
  - Capacity utilization ← can't share unused space
  - Miss rate ← smaller caches

Multilevel Caches

- Primary (L1) caches attached to CPU IMEM, DMEM
  - Small, but fast
  - Focusing on hit time rather than hit rate
- Level-2 cache services misses from primary cache L2
  - Larger, slower, but still faster than main memory
  - Unified instruction and data (why?)
  - Focusing on hit rate rather than hit time (why?)
- Main memory services L-2 cache misses
  - Some high-end systems include L-3 cache

E.G. w/o L_2

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns

1 cycle $\Rightarrow \frac{1}{4G}$ sec $= 0.25$ ns

miss penalty $= 100\text{ ns}\left(\frac{1\text{ cycle}}{\frac{1}{4}\text{ ns}}\right) = 400$ cycles

- With just a primary (L1) cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

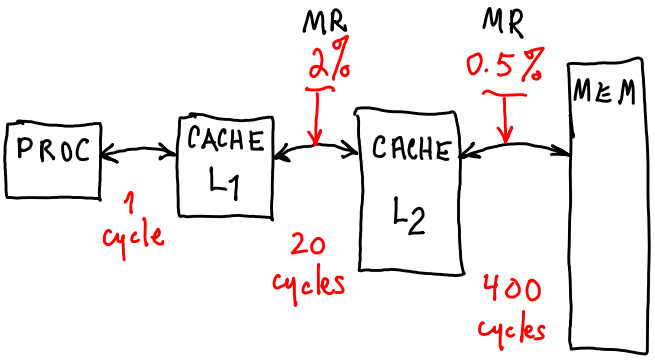
MR

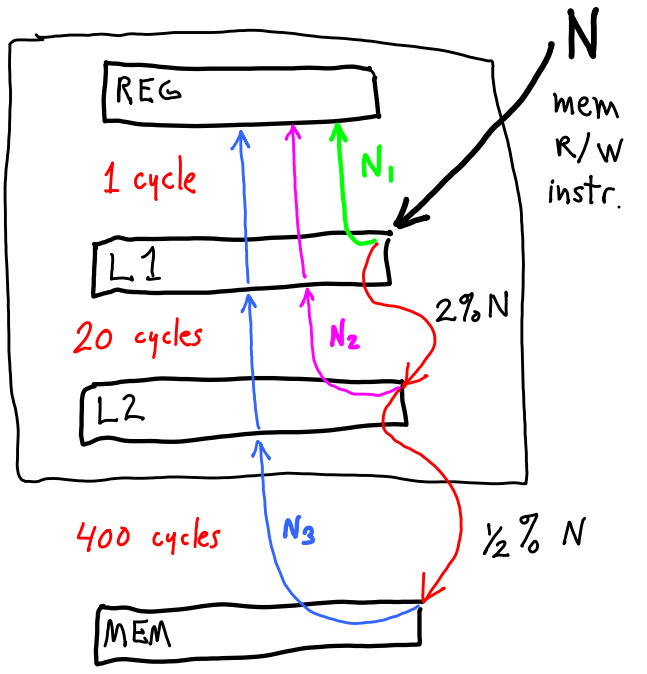
$CPI = (98\%)(1\text{ cycle for hit}) + (2\%)(400\text{ cycle stall} + 1\text{ cycle})$

$= 0.98 + 0.02(400) + 0.02(1) = 1 + 0.02(400) = 9$

E.G. w/ L_2

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6



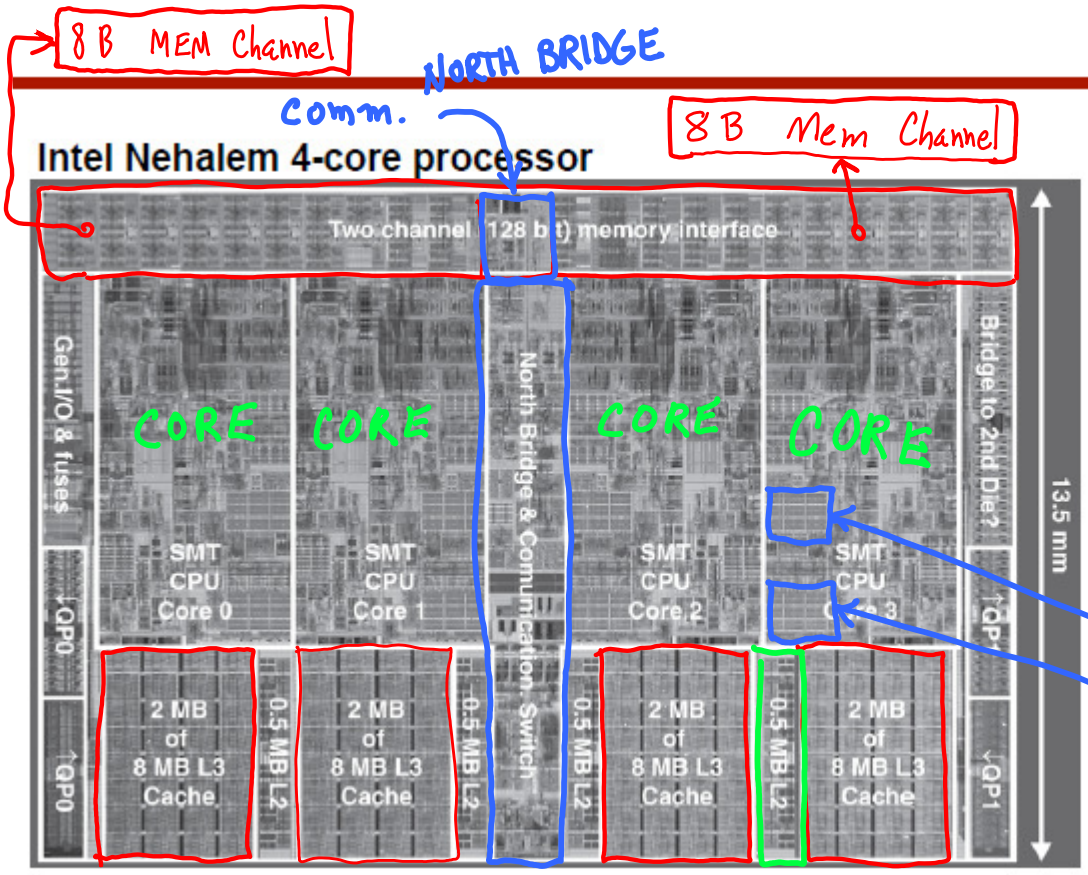


$CPI = \frac{\#\text{ cycles}}{N\text{ instructions}}$

$= (1/N)\left[(N_1 + N_2 + N_3)(1) + (N_2 + N_3)(20) + N_3(400)\right]$

$N_1 = 98\%\,N \quad N_3 = \frac{1}{2}\%\,N \quad N_2 = N - (N_1 + N_3) \Rightarrow (N_2 + N_3) = N - N_1 = 2\%\,N$

$= \left[N(1) + 2\%N(20) + \frac{1}{2}\%N(400)\right]/N = 0.98 + 0.02(20) + 0.005(400) = 3.4$



	Intel Nehalem (P6 Quad)	AMD Opteron X4
L1 caches ($I) (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles
($D)	L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 3 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time 9 cycles
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 38 cycles

hit 3 cycles

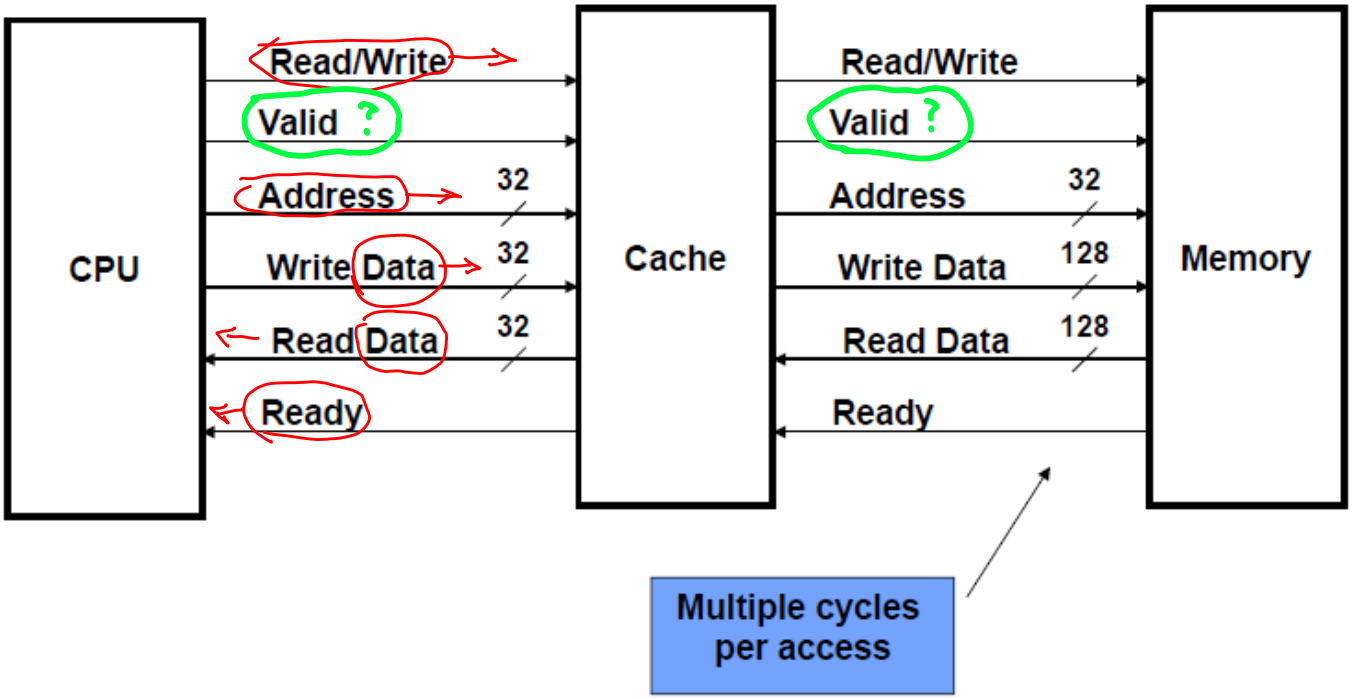
hit 9 cycles

hit 38 cycles

n/a: data not available

64B Blocks = 16 32-bit words
8 64-bit words

Interface Signals

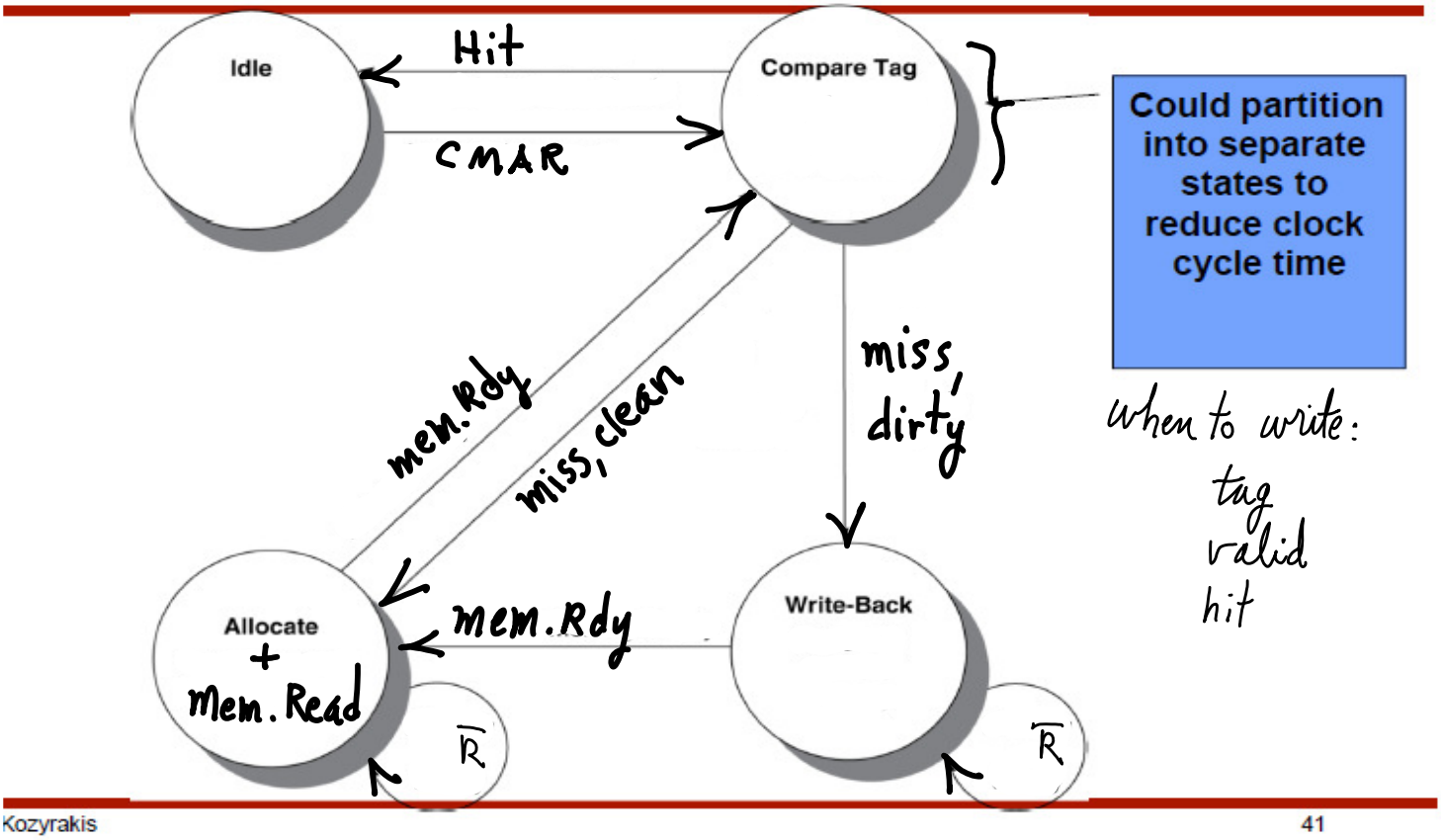


See (LC3-based cache projects):

http://pages.cs.wisc.edu/~karu/courses/cs552/spring2009/wiki/index.php/Main/CacheModule

http://www.ece.ncsu.edu/muse/courses/ece406spr09/labs/proj2/proj2_spr09.pdf

Cache Controller FSM



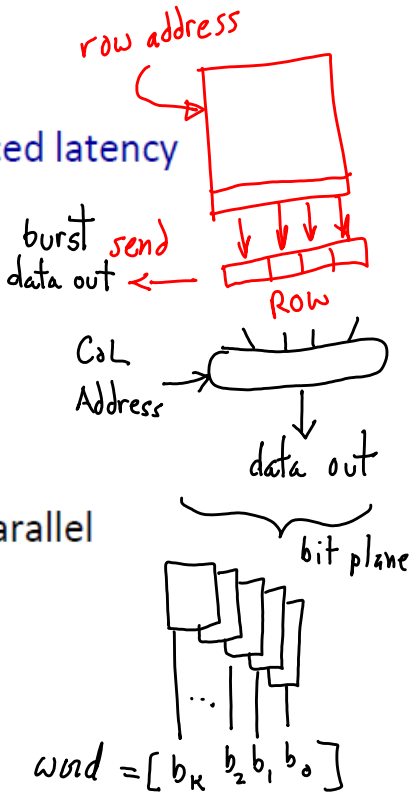
- SRAM
  - Requires low power to retain bit
  - Requires 6 transistors/bit

- DRAM
  - Must be re-written after being read
  - Must also be periodically refeshed
    - Every ~ 8 ms
    - Each row can be refreshed simultaneously
  - One transistor/bit
  - Address lines are multiplexed:
    - Upper half of address: row access strobe (RAS)
    - Lower half of address: column access strobe (CAS)

- Some optimizations:
  - Multiple accesses to same row
  - Synchronous DRAM
    - Added clock to DRAM interface
    - Burst mode with critical word first
  - Wider interfaces
  - Double data rate (DDR)
  - Multiple banks on each DRAM device

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Four transfers per cycle

DDR + 2 data bus (in, out) × 2 clocks

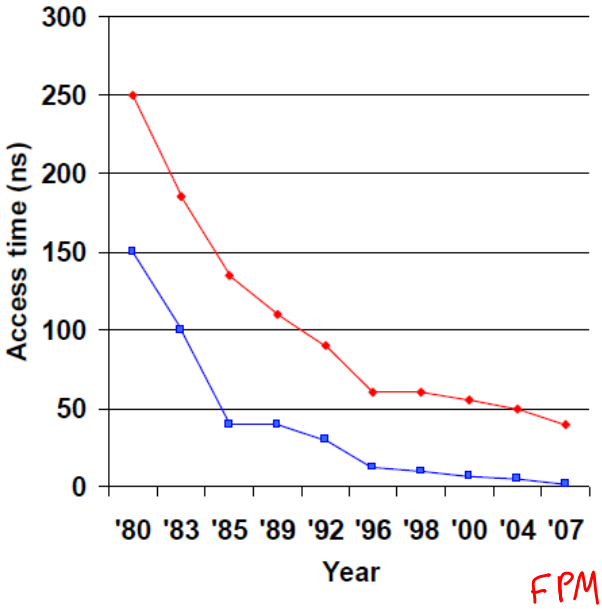


- DIMMs: small boards with multiple DRAM chips connected in parallel
  - Functions as a higher capacity, wider interface DRAM chip
  - Easier to manipulate, replace, ...

			Row access strobe (RAS)			
Production year	Chip size	DRAM Type	Slowest DRAM (ns)	Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

DRAM Generations & Trends

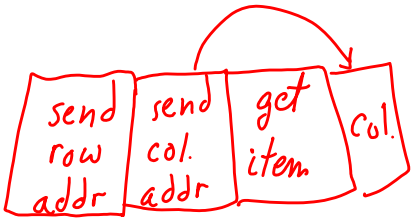
Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



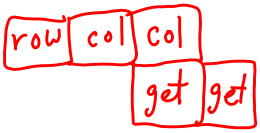
Total Time for Random address

delay per column

FPM



EDO



...

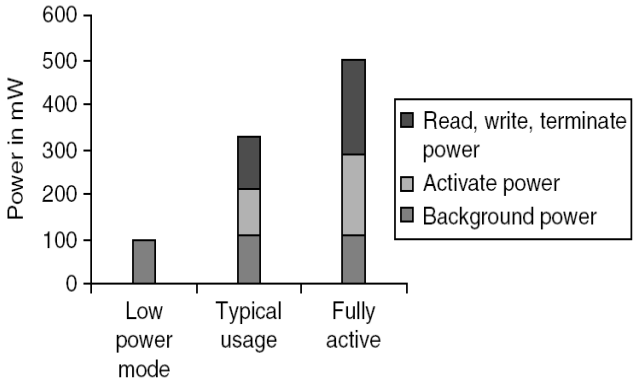
Burst

...

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

- DDR:
  - DDR2
    - Lower power (2.5 V -> 1.8 V)
    - Higher clock rates (266 MHz, 333 MHz, 400 MHz)
  - DDR3
    - 1.5 V
    - 800 MHz
  - DDR4
    - 1-1.2 V
    - 1600 MHz

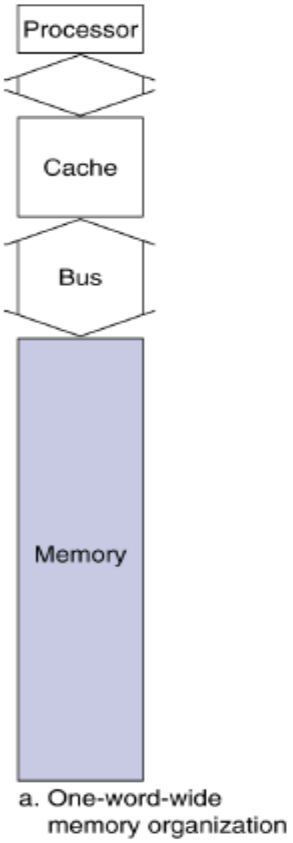
- Graphics memory:
  - Achieve 2-5 X bandwidth per DRAM vs. DDR3
    - Wider interfaces (32 vs. 16 bit)
    - Higher clock rate
      - Possible because they are attached via soldering instead of socketted DIMM modules



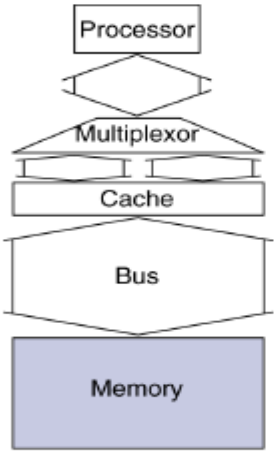
- Memory is susceptible to cosmic rays
- *Soft errors*: dynamic errors
  - Detected and fixed by error correcting codes (ECC)
- *Hard errors*: permanent errors
  - Use sparse rows to replace defective rows

- Chipkill: a RAID-like error recovery technique

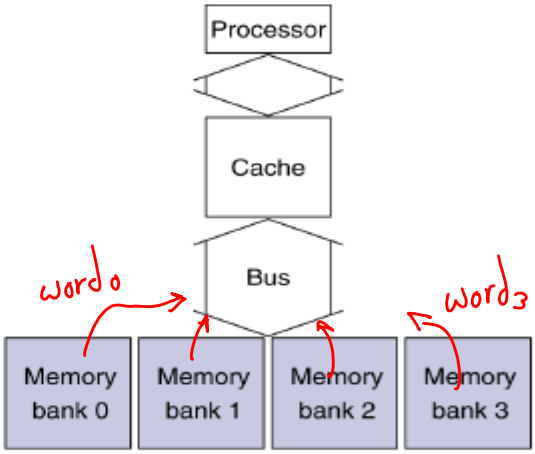
Increasing Memory Bandwidth



a. One-word-wide memory organization



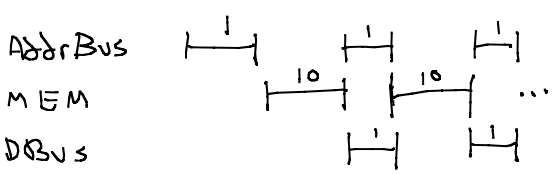
b. Wider memory organization



c. Interleaved memory organization

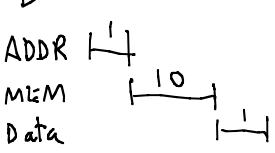
- 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles per 4 words
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4 × 1 = 20 bus cycles per 4 Words
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

A



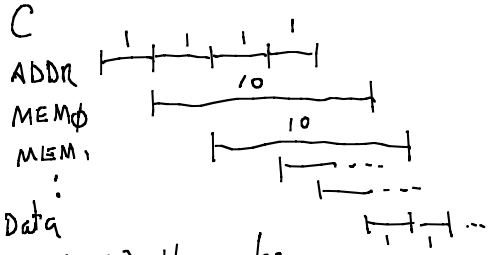
1 + 11.4 cycles / 4 words

B



12 cycles / 4 words

C



1+10+4 cycles / 4 WORDS

- Six basic cache optimizations:
  - Larger block size → spatial locality
    - Reduces compulsory misses
    - Increases capacity and conflict misses, increases miss penalty
  - Larger total cache capacity to reduce miss rate
    - Increases hit time, increases power consumption
  - Higher associativity
    - Reduces conflict misses
    - Increases hit time, increases power consumption
  - Higher number of cache levels
    - Reduces overall memory access time
  - Giving priority to read misses over writes
    - Reduces miss penalty — more reads than writes
  - Avoiding address translation in cache indexing
    - Reduces hit time