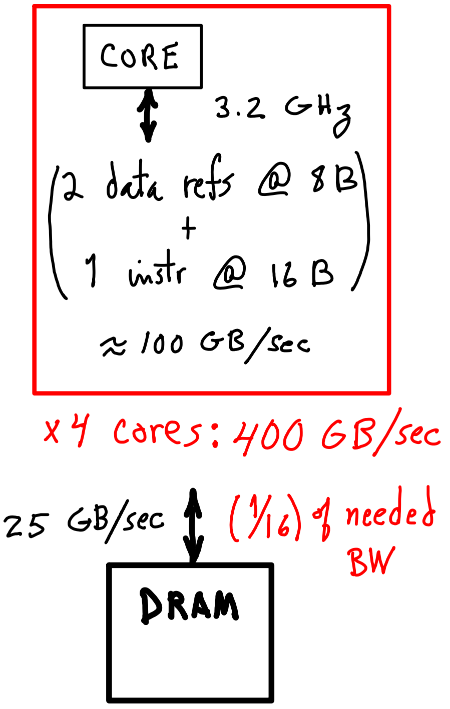
Cache

a problem

- 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

Amdahl, 94%

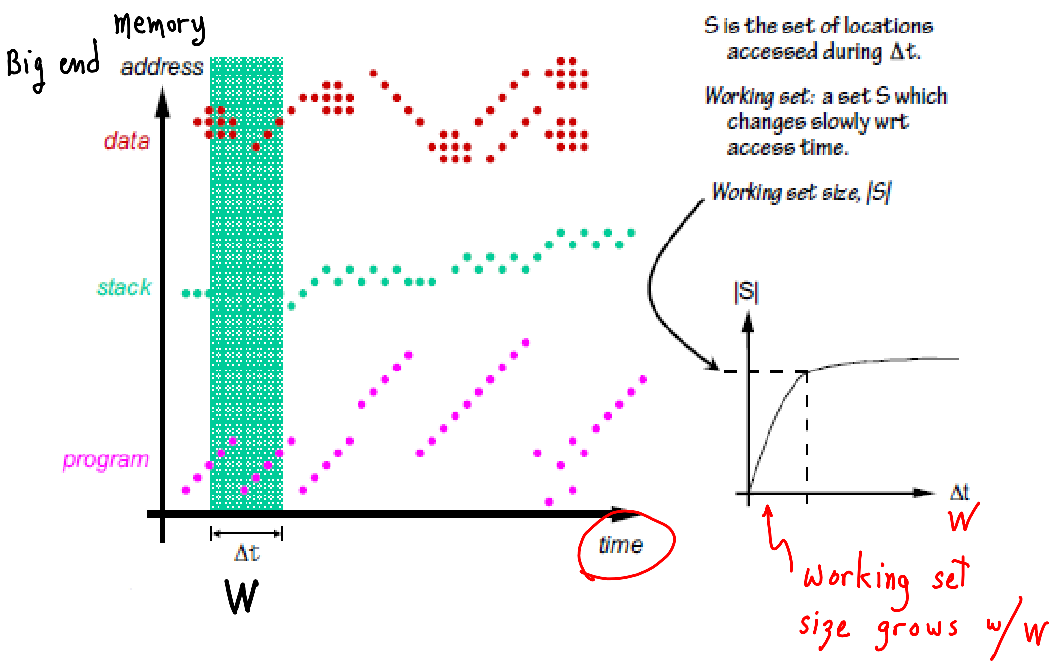


Programs ignore this. Can we help?

Maybe.

Is there Locality ?

Memory Reference Patterns



Pick a **time window size w**.

In time span w, are there,

Multiple References, to nearby addresses: **Spatial Locality**

Repeated References, to a set of locations: **Temporal Locality**

Take advantage of behavior patterns.

If stable patterns last, Long Enough (?)

Size of Locality depends on W

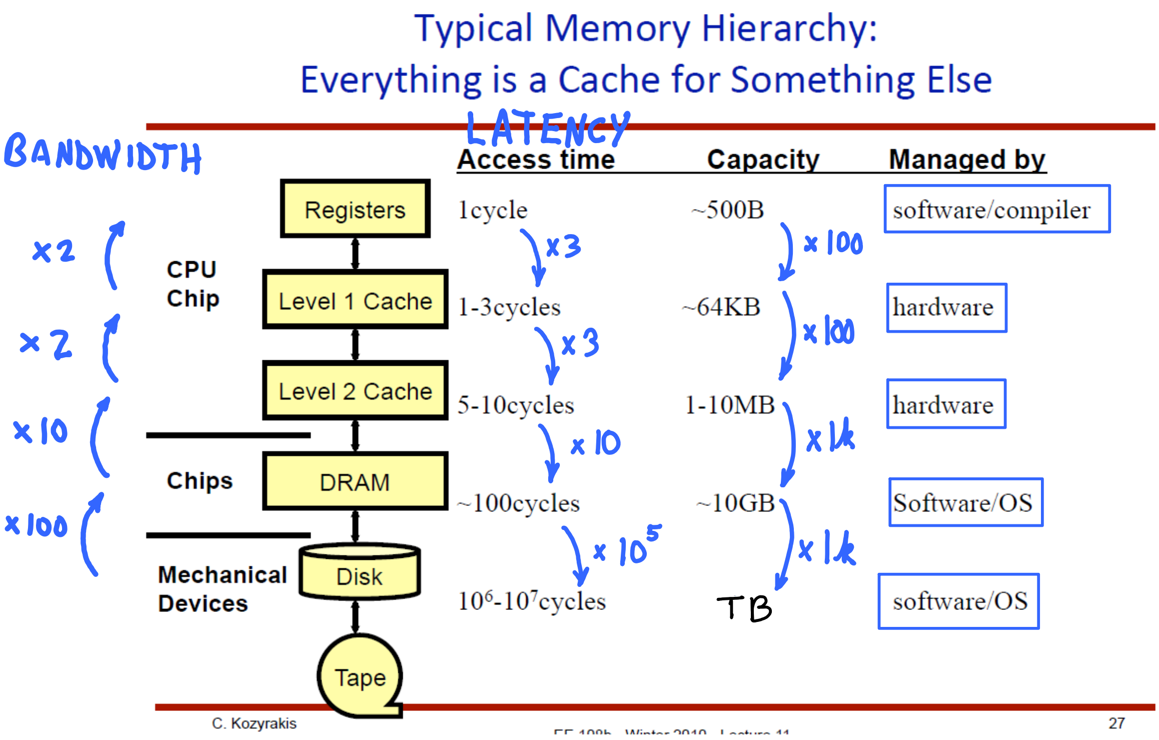
W ==> **total execution time**, **everything** is local

W ==> **one instruction time**, **single address** is local

Trade-off

Short time ===> **Small** set

Long time ===> **Large** set



Typical Memory Hierarchy: Everything is a Cache for Something Else

	Access time	Capacity	Managed by
Registers	1cycle	~500B	software/compiler
Level 1 Cache	1-3cycles	~64KB	hardware
Level 2 Cache	5-10cycles	1-10MB	hardware
DRAM	~100cycles	~10GB	Software/OS
Disk	10^6-10^7cycles	TB	software/OS
Tape			

C. Kozyrakis

EE 108b Winter 2010 Lecture 11

27

(BANDWIDTH B/sec ↓)

and

(LATENCY cycles ↑)

Larger gap in access time.

therefore do what?

Hide latency.

using what?

Technology Tradeoffs

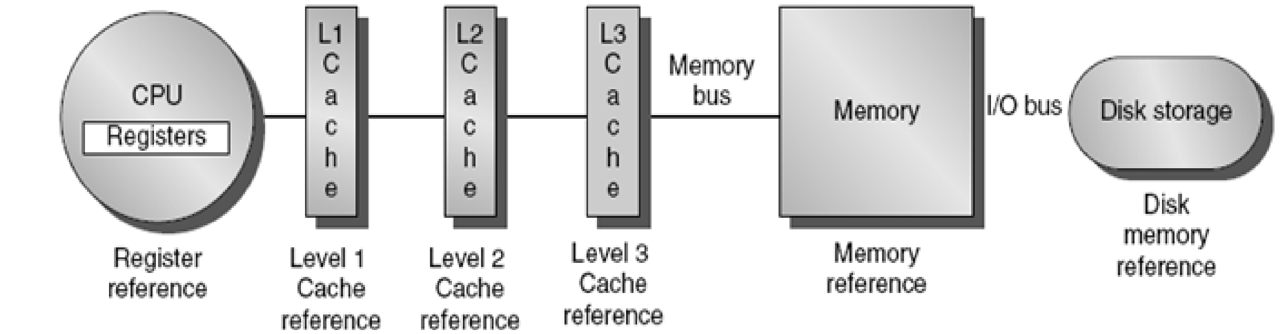
Large set, Many bits ===> Bad: (Bandwidth, Latency), Good: ($, Area, Watts) per bit

Small set, Few bits ===> Good: (Bandwidth, Latency), Bad: ($, Area, Watts) per bit

Small w → fast set turn over → more bandwidth (low latency)

large w → slow set turn over → less bandwidth (high latency)

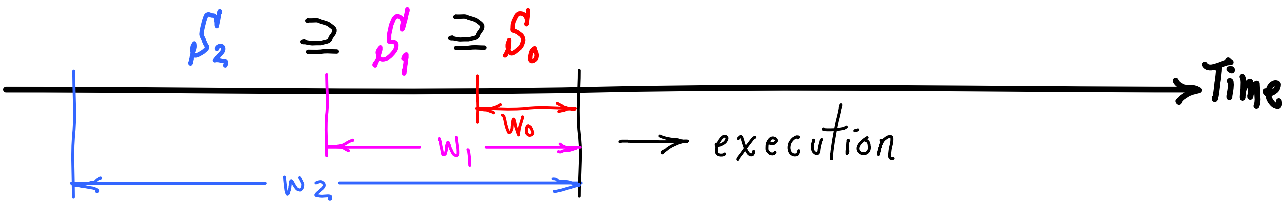
$W_0 < W_1 < W_2 < W_3 < W_4 < W_5$



	Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference	Memory reference	Disk memory reference
Size:	1000 bytes	64 KB	256 KB	2–4 MB	4–16 GB	4–16 TB
Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	5–10 ms

capacity
latency

(a) Memory hierarchy for server



We hope

Most **changes** in S_0 **refer to** items in S_1

Most **changes** in S_1 **refer to** items in S_2

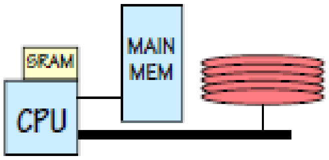
etc. ...

↓ less bandwidth required
latency overlapped or hidden

Exploiting the Memory Hierarchy

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

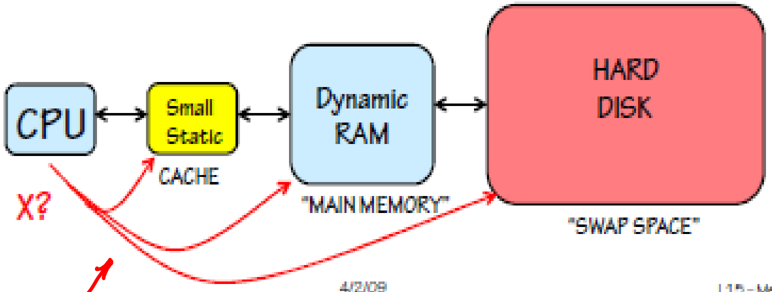
- Registers, Main Memory, Disk each available as storage alternatives;
- Tell programmers: "Use them cleverly"



Programs do manage cache effects

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.



Programs do not take into account cache effects, hope for the best.

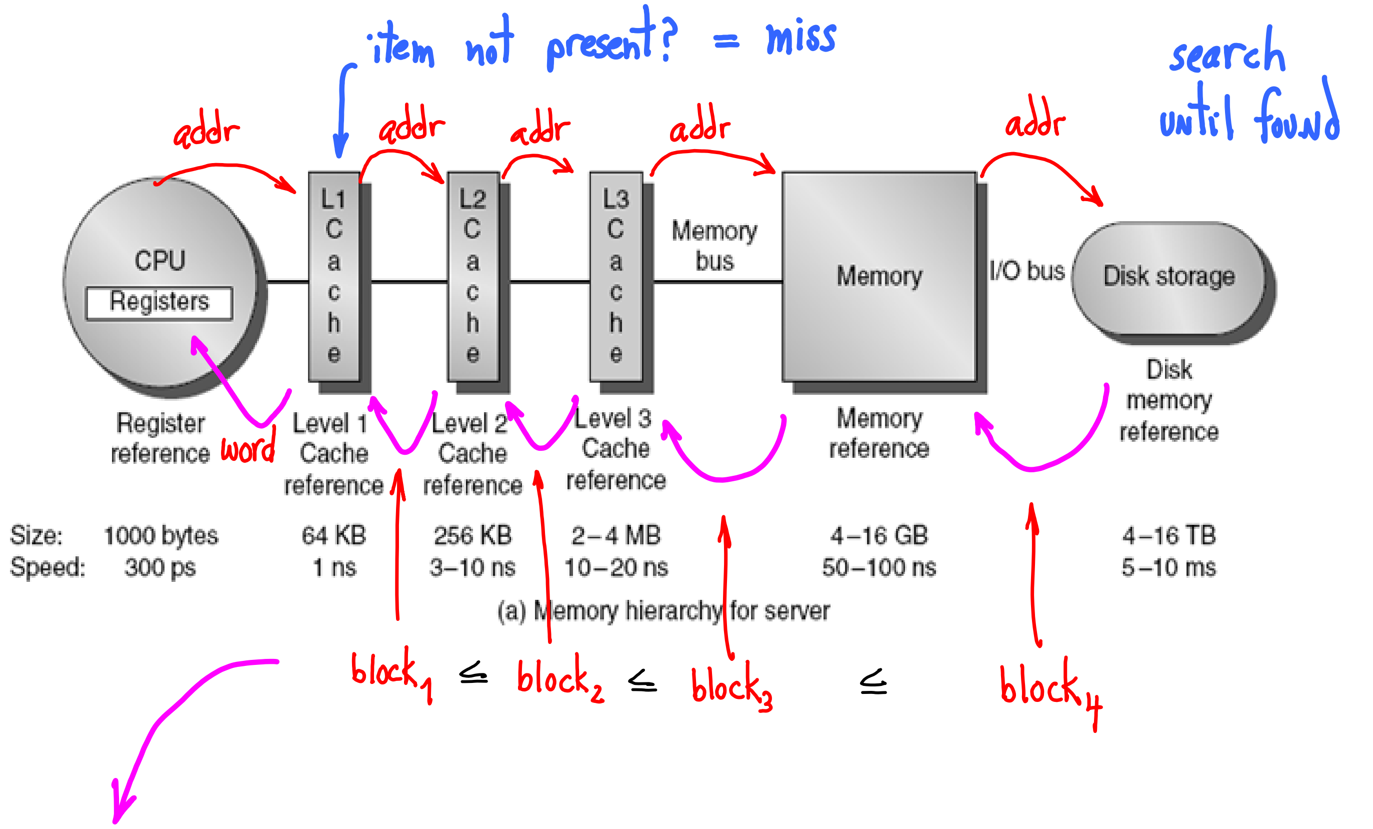
HW and system SW manage moving data

e.g.

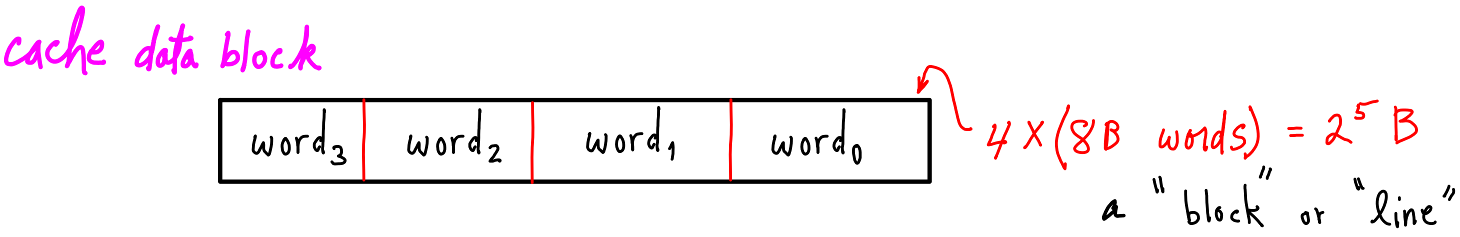
register loading / unloading : compiler

L1, L2, L3 : cache controllers

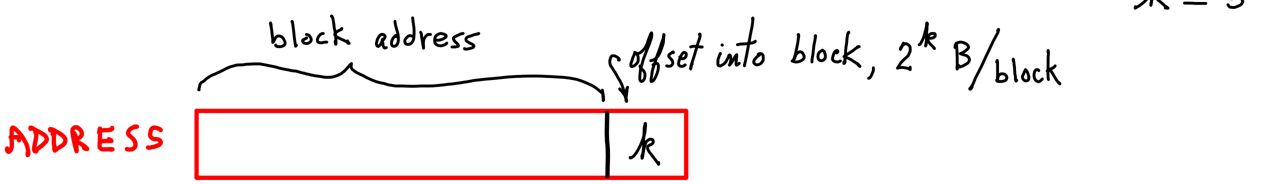
Memory / disk : OS software, disk controllers



(a) Memory hierarchy for server



$k = 5$



Transfer a block at a time:

--- latency for 1-st word

--- remainder at bandwidth rate, hopefully

Block size varies from level to level (2X)

--- Pay delay for block transfer, but what if other words never used?

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

$$MR_i = N_{miss} / N_{access}$$

$(miss_i = \text{not found in level } i)$

- Causes of misses
  - Compulsory
    - First reference to a block ⇒ no choice, 1st reference (? prefetch)
  - Capacity
    - Blocks discarded and later retrieved ⇒ couldn't keep in cache, but wanted to
  - Conflict
    - Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache ⇒ cache storage scheme fault

(+ Coherency + context switching)

$$HR = (1 - MR)$$
$$= N_{hit} / N_{access}$$

Metrics:

$$\text{Avg Access time} = (\text{hit rate})(\text{hit time}) + (\text{miss rate})(\text{miss time})$$
$$\text{Avg Power} = \#(\text{active devices})(\text{avg dynamic power})$$

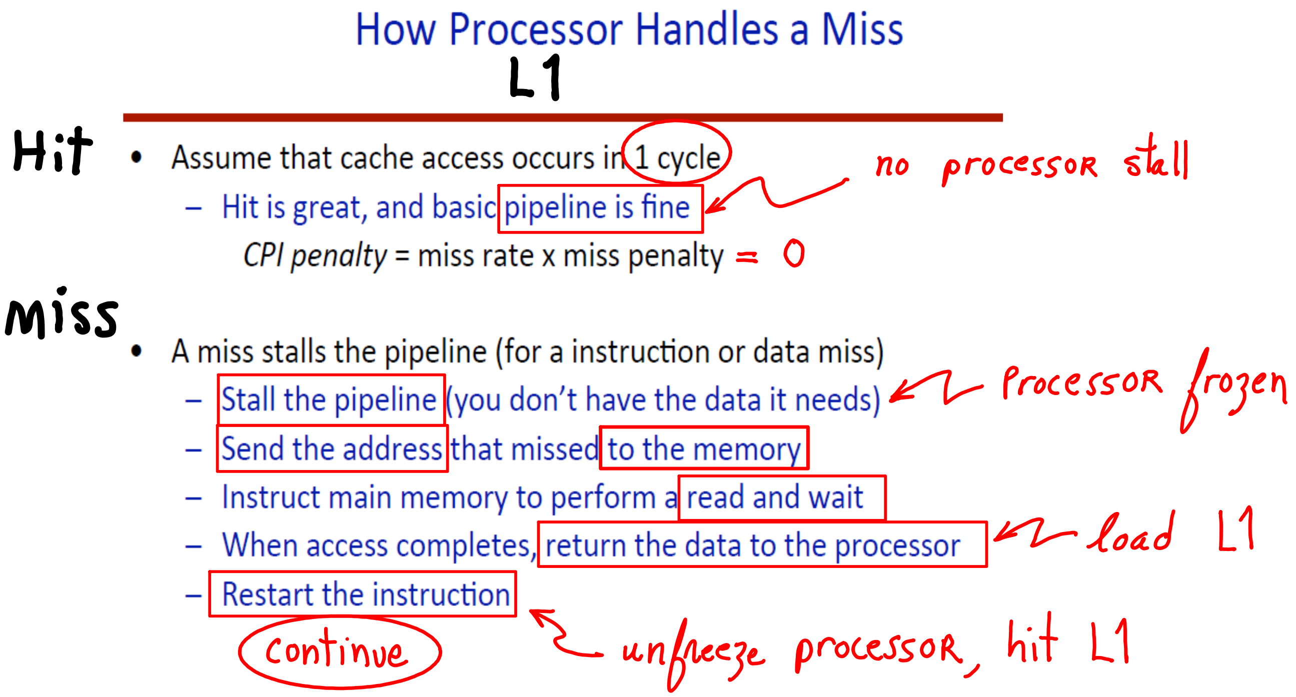
$$\begin{aligned}
\text{AMAT} &= (\overbrace{\text{hit rate}}^{\%\ \text{hits}})(\text{hit time}) + (\text{miss rate})(\overbrace{\text{miss time}}^{\text{time to R/W data + cache hit time}}) \\
&= (1-MR)\, T_{hit} + MR(T_{access} + T_{hit}) \\
&= ((1-MR) + MR)\, T_{hit} + MR\, T_{access} \\
&= T_{hit} + MR(\underbrace{T_{access}}_{\text{miss Penalty}})
\end{aligned}$$

AMAT can be w.r.t.

Global performance
or
Level i performance

What's important?
Overall performance = **execution time**
or
= **average CPI**

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



How Processor Handles a Miss

L1

Hit

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

    CPI penalty = miss rate x miss penalty = 0

no processor stall

Miss

- 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

Processor frozen

load L1

continue

unfreeze processor, hit L1

We can Generalize

A Turing Machine Tape

R/W head moves L or R, copy a region at a time.

Cost is proportional to distance and size of region copied.

Cache Organization and Methods

--- **Big Memory**, **Small Cache** ===> **Block Mapping**
(how to place blocks in cache)

Associative: **anything goes** anywhere, check contents (contains address)
complex + expensive (area, power)

Direct Mapped: (**like a Reg File**, but words are blocks)
simple + fast, but too restrictive placement?

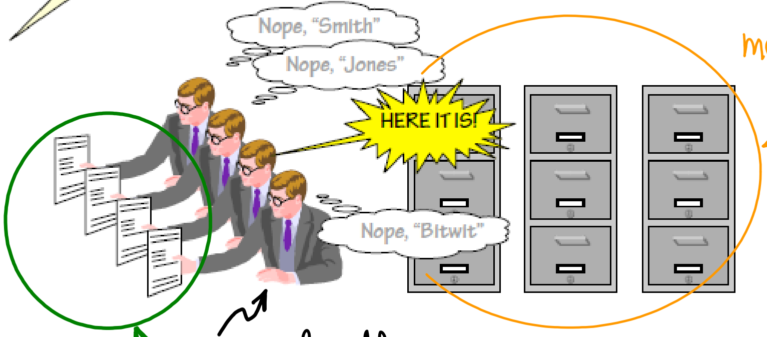
Set Associative: (hybrid of Associative and Direct Mapped)

Some Block Parameters

--- **How big**? Spatial locality captured by fetching neighboring data/instructions.
--- **Replace** what when? Working set captures temporal locality.
--- **Writing**, when, where? Change locally or globally, maintain correct program behavior.

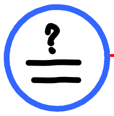
Associativity: Parallel Lookup

Find "Bitdiddle, Ben" ⟷ request's address, e.g., lw R1, R2, #10

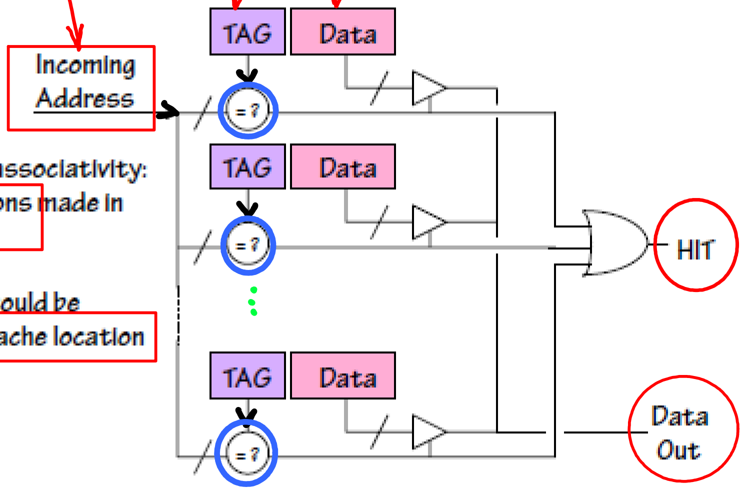


search all in parallel

cache contents



compare → match?



The extreme in associativity: All comparisons made in parallel

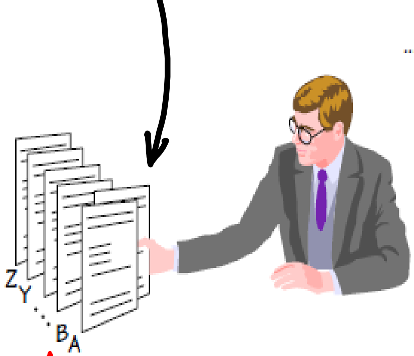
Any data item could be located in any cache location

item can be in any slot; load new item into any empty.

Direct-Mapped Cache

(non-associative)

Find "Bitdiddle, Ben"



slot for "B"s

NO Parallelism:

Look in JUST ONE place, determined by parameters of incoming request (address bits)

... can use ordinary RAM as table

Need: Address Mapping Function!

Maps incoming BIG address to small CACHE address… tells us which *single* cache location to use

Direct Mapped: just use a subset of incoming address bits!

Collision when several addresses map to same cache line.

cannot cache both "Bitdiddle" and "Bytetwaddle"

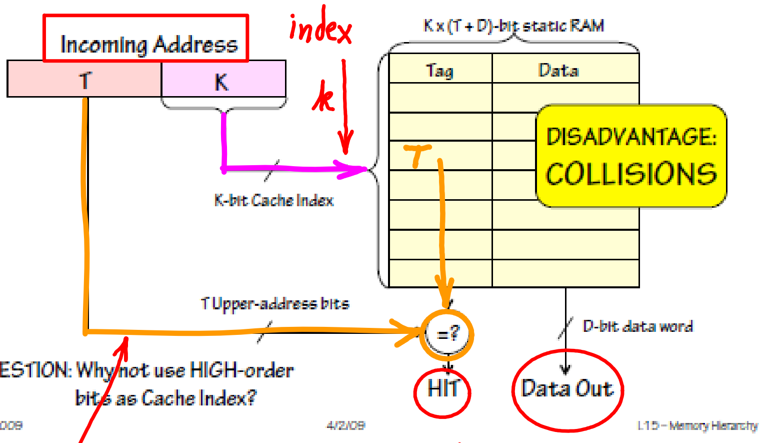
Direct Mapped Cache

≅ Reg file

Low-cost extreme:

Single comparator

Use ordinary (fast) static RAM for cache tags & data:

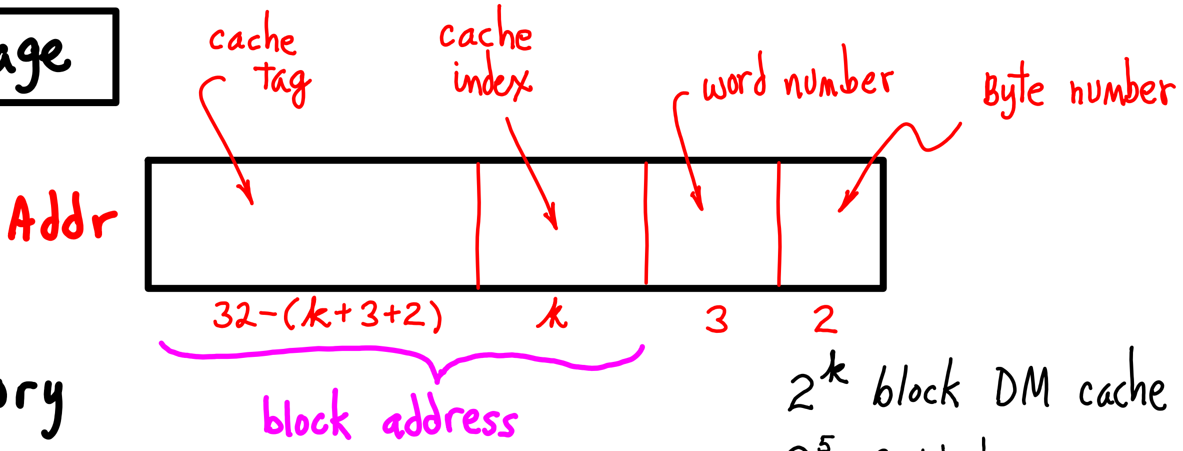


QUESTION: Why not use HIGH-order bits as Cache Index?

only need to use part of address

Address bit usage

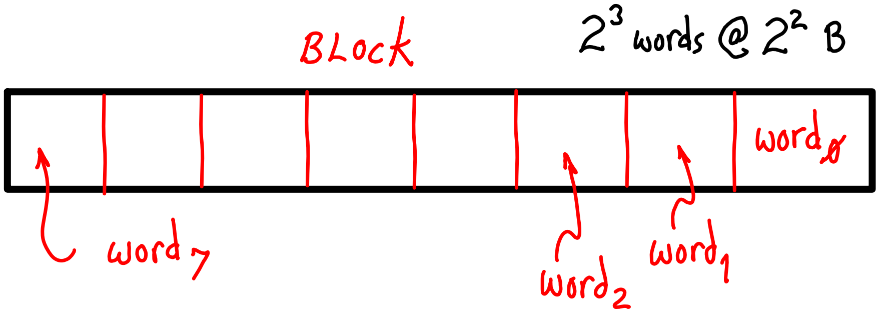
Byte addressable
4-Byte words
8-word blocks



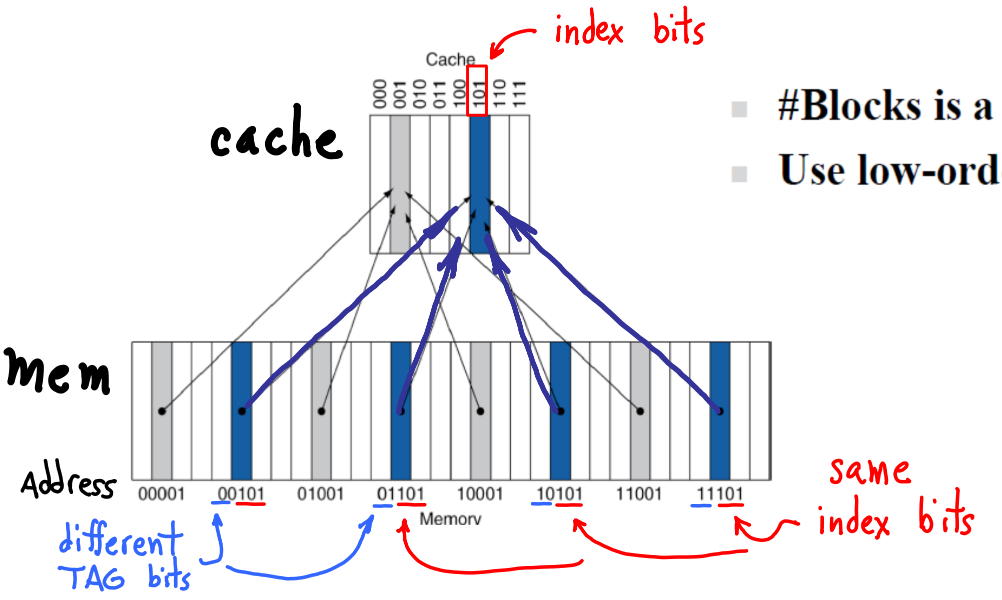
2^k block DM cache
2^5 B blocks
2^3 words @ 2^2 B

word address	Memory	
00 ... 0000	$B_3\ B_2\ B_1\ B_0$	word 0
00 ... 0100	$B_3\ B_2\ B_1\ B_0$	word 1
00 ... 1000	$B_3\ B_2\ B_1\ B_0$	word 2

00 ... 1011 Byte address



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



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

Collisions spread by 2^3
⇒ less spatially local

- 8-blocks,
- Initial state

We use TAG bits to identify which block.

But, what about at startup?

--- Content is random

--- boot process initializes valid bit (V = 0)

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

Example (ignore block and byte offset bits)
DM, 3-bit index

LC3 assembly:
LW R1, R2, #0

MIPS assembly:
L $1, 0($2)

10110 R2

MAR

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

Compulsory/Cold Miss

data cache?
instruction cache?
mixed cache?

index

Index	V	Tag	Data or instruction
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

not Valid
Stall,
Access Memory
Write Cache (tag + data)
Set Valid bit
Restart Instruction

LC3 assembly:
LW R3, R4, #0

R4 11010

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
STALL, FETCH
WRITE, RESTART

LC3 assembly:
SW R5, R4, #0

R4 11010

≟

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

Hit

5678 R5

write
To cache

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]
111	N		

VALID = 1
+
TAGs match
Write data to cache
Write data to memory
(when?)

LC3 assembly:
SW R7, R6, #0

R6 10010

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

Replacement

12BF R7

write new tag

write data

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010] ~~5678~~
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

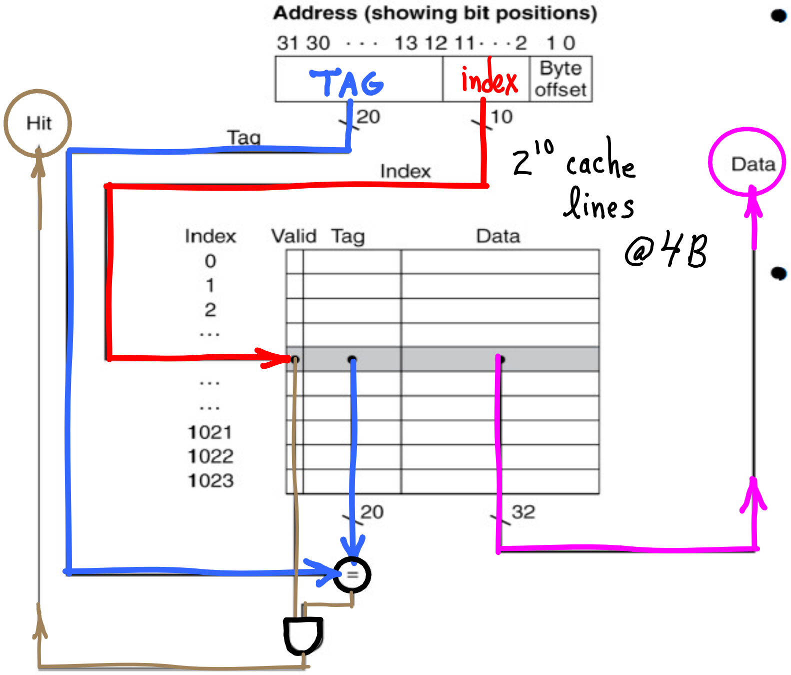
VALID = 1
+
TAGs do not match
= Collision

Stall, write old data to mem
write data to cache
write tag to cache

write new data to mem?

Example:

DM, 32-bit address, byte-addressable, 1-word blocks (32-bit word = 4-byte block)



- Assumptions
  - 32-bit address
  - 4 Kbyte cache
  - 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

Need only compare upper 20 bits as tag, index bits are the same for any item in same slot.

diff. Same
Tags index

LW R1, < address = 1100110 >
LW R2, < address = 0101110 >
SW R3, < address = 1100110 >
SW R4, < address = 0101110 >
LW R5, < address = 1100110 >

⟹ Thrashing

Each access evicts something needed later, or causes a miss.

Worse than no cache!

Can happen at any level or type of caching:

Direct Mapped, Conflicts (as above)

Fully Associative, Capacity
e.g., Virtual Memory Page Thrashing

- Consider the following example code:

Cache

1k blocks
@ 8B

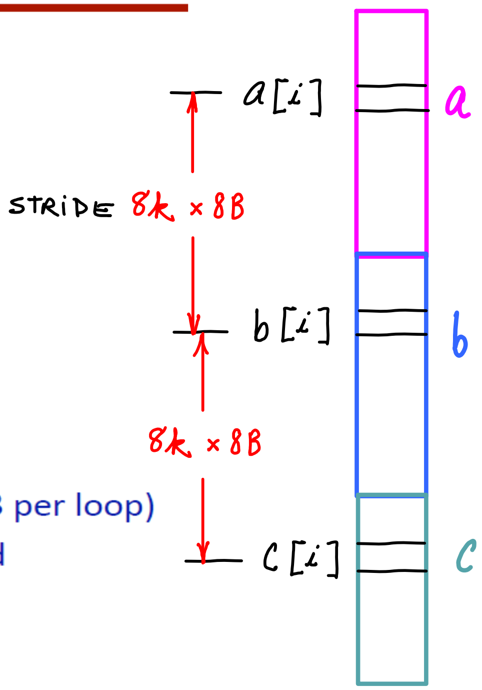
→ 2^{10} blocks

→ 10-bit index

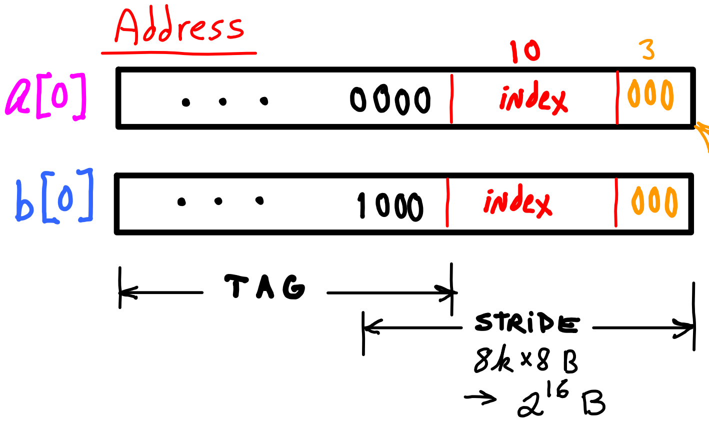
```
double a[8192], b[8192], c[8192];
void vector_sum()
{
     int i;

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

8k × 64b 8k × 8B 8k × 8B

  - 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?



Stride in **multiples of 2^13 :**
===> indices same,
tags differ.



8B block
→ 3-bit offset
64-bit Double
→ 3 bit offset

How can we fix this?

Bigger cache? How big? → index + offset > 17 bits (recall, c also)

→ size ≥ 2^{18} B = 256 kB, 15-bit index

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

How much is the programmer responsible for?

Portable code, different architectures?

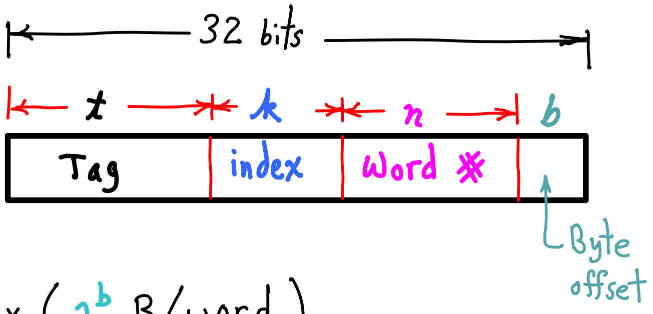
Irregular data layouts a solution?

Compiler's responsibility?

Block Size Effects

8 kB Cache

Address



$$\Longrightarrow (2^{k} \text{ blocks}) \times (2^{n} \text{ words/block}) \times (2^{b} \text{ B/word})$$

Each cache line = [tag bits] [data block bits]

Total cache size = (# lines) X (# tag bits + # data bits)

Storage overhead = (total # tag bits) / (total # data bits)

$$(2^{10} \text{ blocks}) \times (1 \text{ word/block}) \times (8 \text{ B/word})$$

(1 latency + 1 Transfer)/word

$k = 10$ $\quad n = 0$ $\quad b = 3$

$$\Longrightarrow t = 32 - (10 + 0 + 3)$$

$$= 19 \text{ bits}$$

$$\Longrightarrow 19/(2^{6} \text{ bits/block}) \cong 1/3 \text{ overhead}$$

vs.

$$(2^{6} \text{ blocks}) \times (16 \text{ words/block}) \times (8 \text{ B/word})$$

$$\left(\frac{1 \text{ latency} + 16 \text{ Transfers}}{16 \text{ words}} \right)$$

$k = 6$ $\quad n = 4$ $\quad b = 3$

$$\Longrightarrow t = 32 - (6 + 4 + 3)$$

$$= 19 \text{ bits}$$

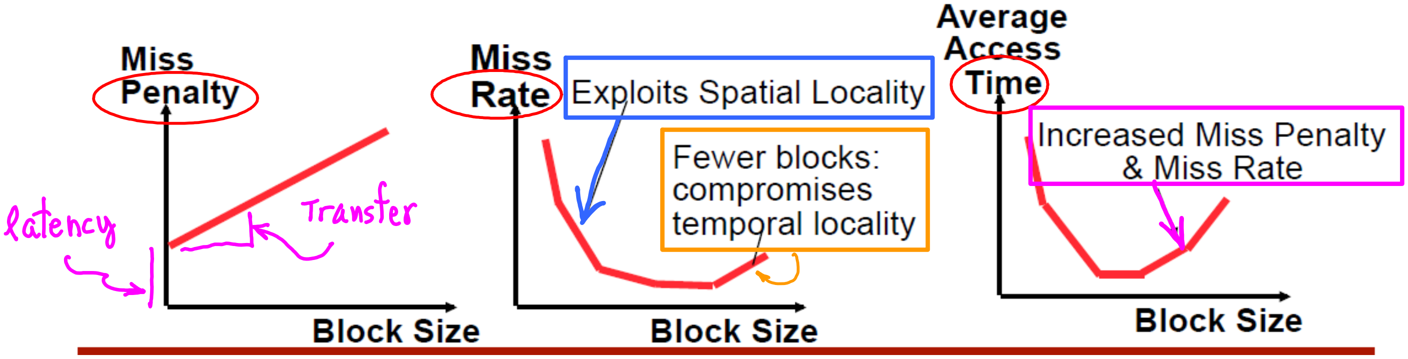
$$\Longrightarrow 19/(2^{4} \times 2^{6} \text{ bits/block}) = 19/1024 \cong 1/50 \text{ overhead}$$

Amortized latency per word
===> 1 / 16

if spatial locality

Block Size vs. Performance

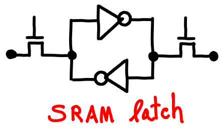
- Larger block sizes take advantage of spatial locality
  - Also incurs larger miss penalty since it takes longer to transfer the block into the cache
  - Large block can also increase the average time or the miss rate
- Tradeoff in selecting block size
- Average Access Time = Hit Time • (1-MR) + Miss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



Averaged over selection of programs: Your performance may be different.

Assume fixed { Bandwidth to memory
Total cache data size → # Blocks = $\frac{\text{Total cache data size}}{\text{Block size}}$

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!

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!

Cost vs Contention

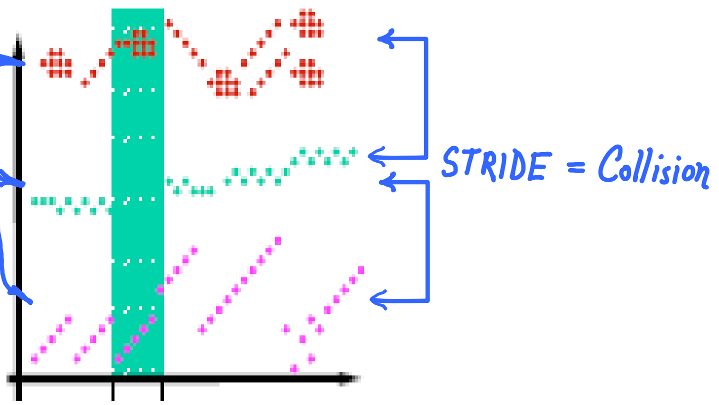
two observations...

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

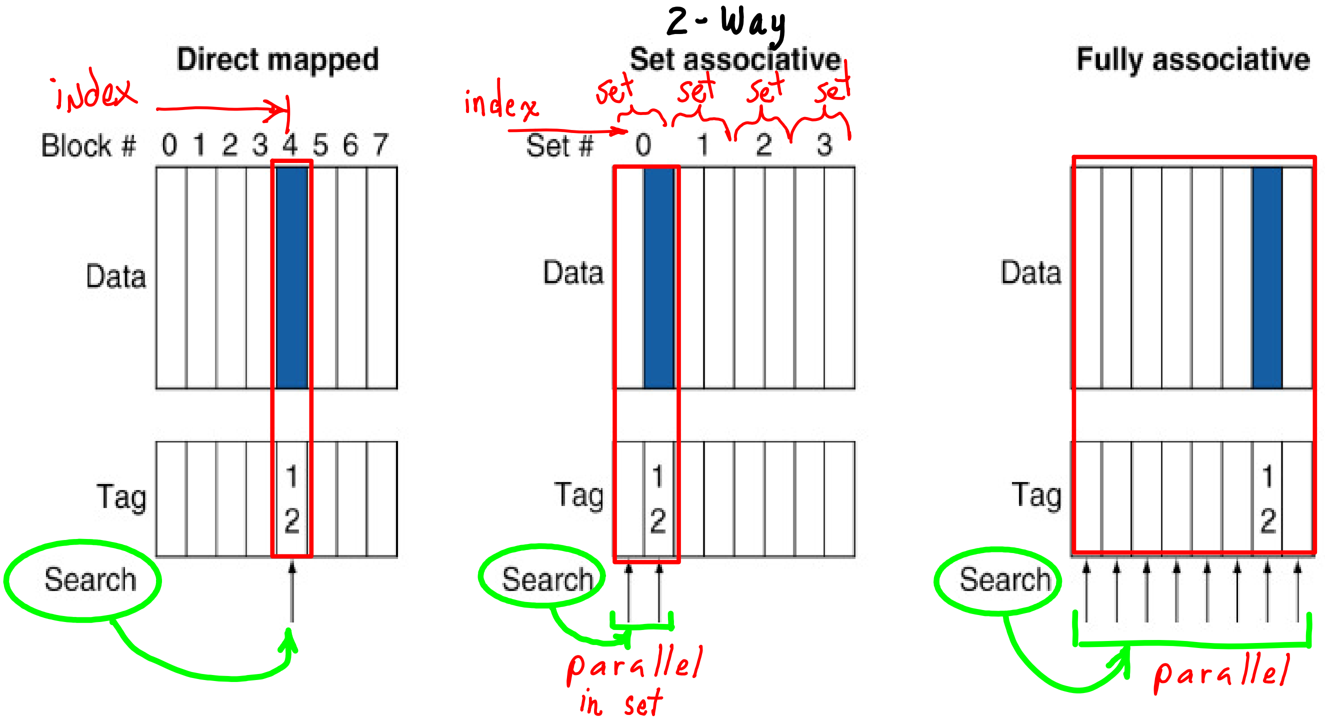
   ... so lets build caches that allow each location to be stored in some restricted set of cache lines, rather than in exactly one (direct mapped) or every line (fully associative).



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

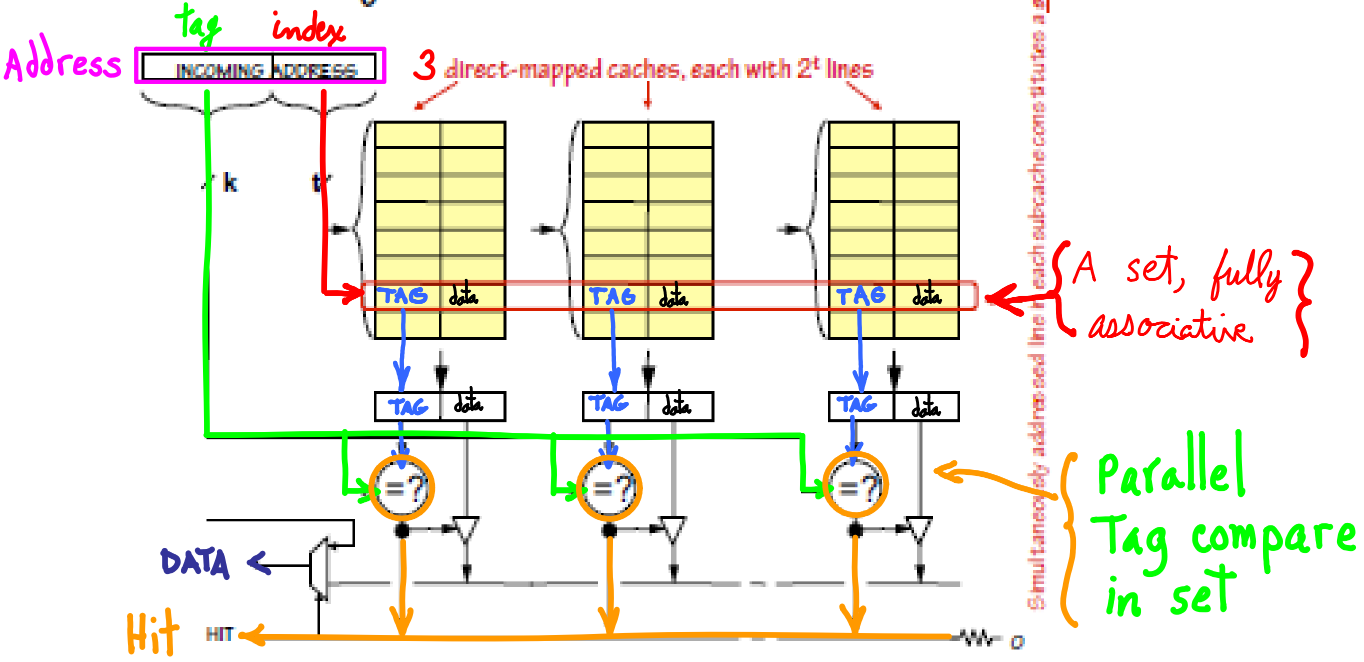
Set Associative Cache



3 Direct Mapped caches → 3-way Associative

8-cache-line, DM caches → 8 sets

3-way Set-Associative Cache



E.G.

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

- Direct mapped

ADDRESS	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0 00	0	miss	Mem[0]			
8 00	0	miss	Mem[8]			
0 00	0	miss	Mem[0]			
6 10	2	miss	Mem[0]		Mem[6]	
8 00	0	miss	Mem[8]		Mem[6]	

4-block cache at t = 0

Collision

collision

Collision

Time

tag

2-bit index

- 2-way set associative

ADDRESS	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0 00	0	miss	Mem[0]			
8 00	0	miss	Mem[0]	Mem[8]		
0 00	0	hit	Mem[0]	Mem[8]		
6 10	0	miss	Mem[0]	Mem[6]		
8 00	0	miss	Mem[8]	Mem[6]		

LRU

TIME

tag

1-bit index

- Fully associative

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

no index

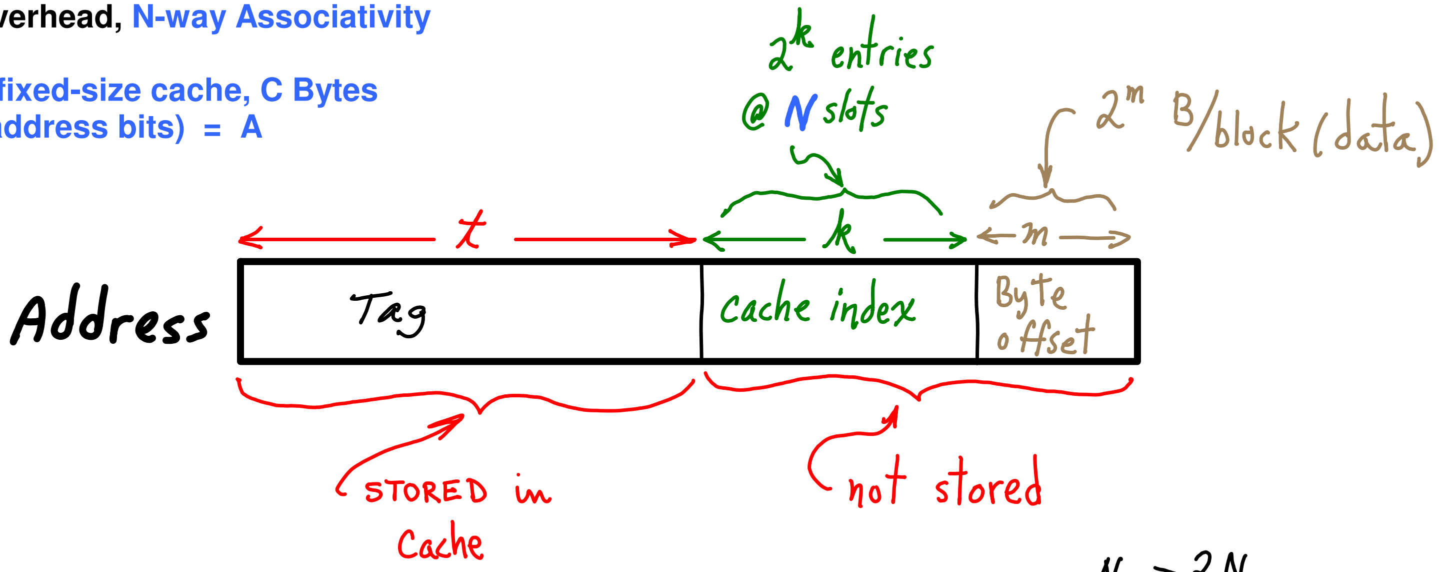
any block can be used

associativity higher ===> tags bigger (overhead?)

Space Overhead, N-way Associativity

Assume fixed-size cache, C Bytes

Total # (address bits) = A



$$C \text{ Bytes} = (\text{tag bits} + \text{data bits})$$

$$= \underbrace{(t \times 2^k \times N)}_{\text{Overhead}} + (2^k \times N \times 2^m \times 2^3)$$

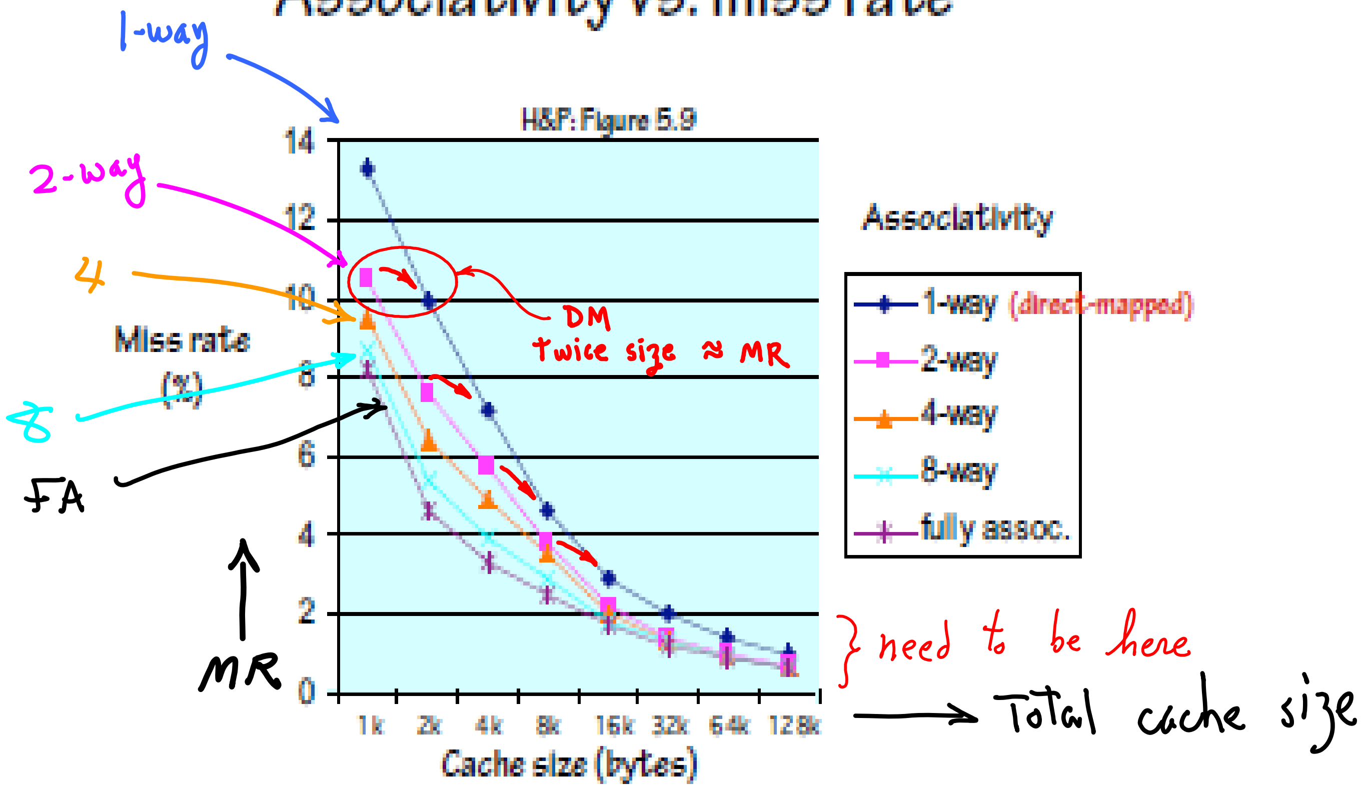
$N \rightarrow 2N$

$k \rightarrow (k-1)$

$t \rightarrow (t+1)$

$t + k + m = A$

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.

A different Job mix

- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000

MR
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

↓ diminishing returns ? 2.5% improvement, is that significant?

What's the metric?

Compare (MR X Miss Penalty) == actual improvement

performance / $?

If $ increment is small ==> bigger N.

$$\frac{T_h(1-MR)+MR\,T_p}{T_h(1-xMR)+xMR\,T_p}$$

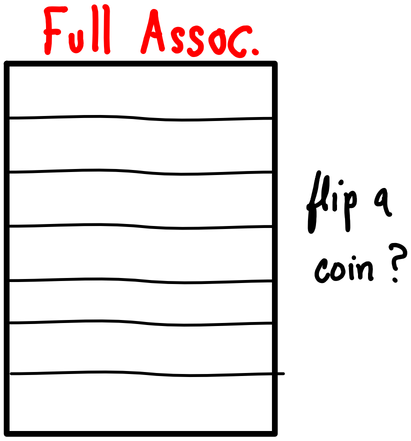
$$= \frac{T_h + MR(T_p-T_h)}{T_h + xMR(T_p-T_h)}$$

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) oldest
    - Often difficult to calculate, so people use approximations. Often they are really not recently used wasn't used since last I looked



Handling of WRITES

What's our workload?

--- How many READS
--- How many WRITES
--- How many READS after WRITES

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".

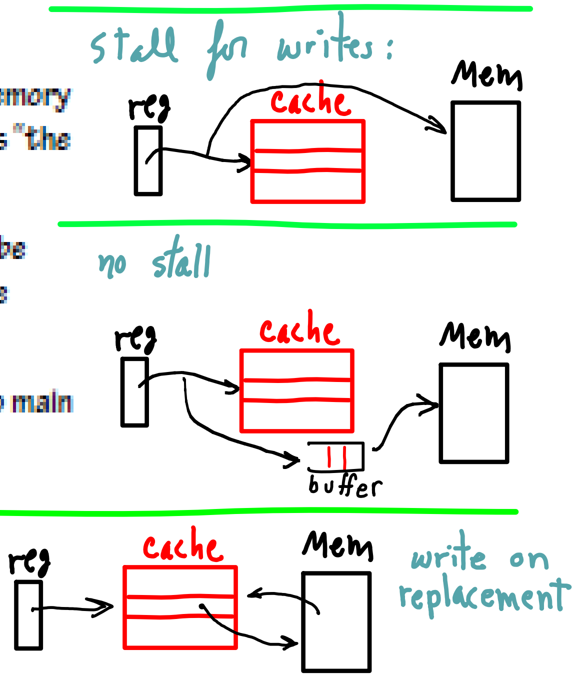
Write-behind: CPU writes are cached; writes to main memory may be buffered, perhaps pipelined. CPU keeps executing while writes are completed (in order) in the background.

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?

no stall until write-back



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

Write Through

Replacement: easy, clobber line (memory always updated → consistent)

Memory Bandwidth: high, every write (as if not using cache)
but only 1-word writes

Processor: stalls on every write

simple, cheap

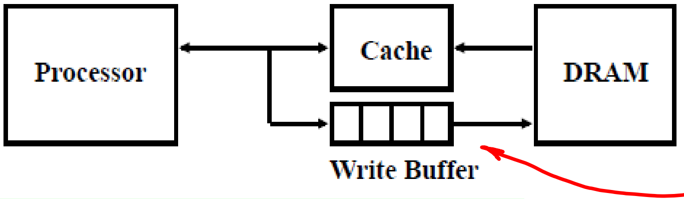
Write Back

Memory inconsistent until replacement (but, multi-processors?)

need dirty bit

Memory Bandwidth: lower load, multiple writes to cache block
but n-word writes (blocks)
but block-write pipelined, efficient

Processor: stalls for write only when dirty block replaced

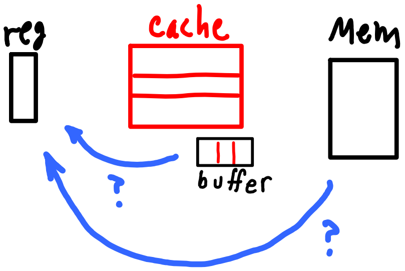


use for either
Write-Back or
Write-Behind

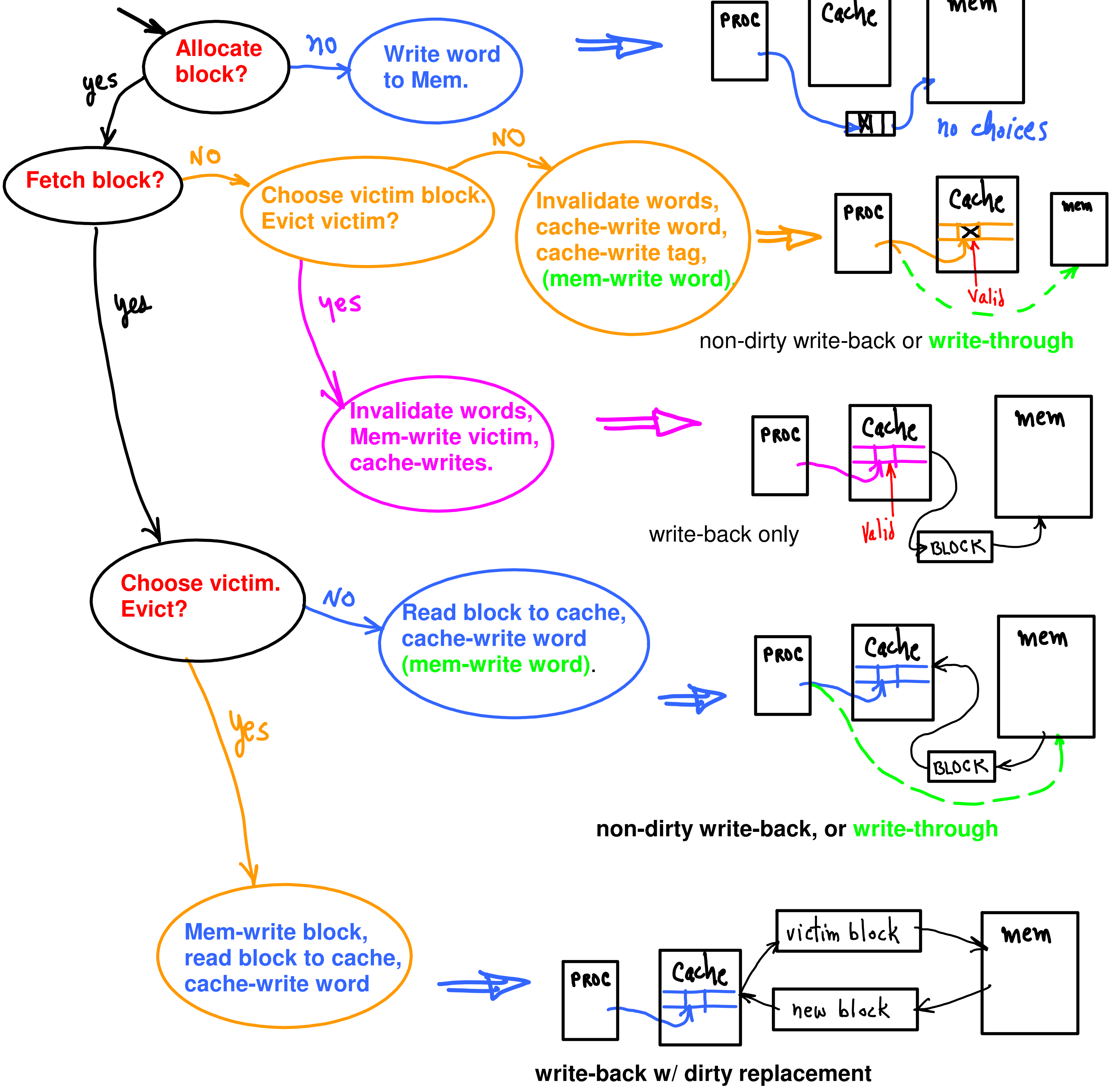
- Use Write Buffer between cache and memory
  - Processor writes data into the cache and the write buffer
  - Memory controller slowly "drains" buffer to memory
- Write Buffer: a first-in-first-out buffer (FIFO)
  - Typically holds a small number of writes
  - Can absorb small bursts as long as the long term rate of writing to the buffer does not exceed the maximum rate of writing to DRAM

write-through w/ buffer, Read Miss?

Where should we look for data?
--- in buffer?
--- in memory?
--- how do we search buffer? Stall if not empty?



what to do on write miss?



Allocate block?

Write word to Mem.

Fetch block?

Choose victim block. Evict victim?

Invalidate words, cache-write word, cache-write tag, (mem-write word).

non-dirty write-back or write-through

Invalidate words, Mem-write victim, cache-writes.

write-back only

Choose victim. Evict?

Read block to cache, cache-write word (mem-write word).

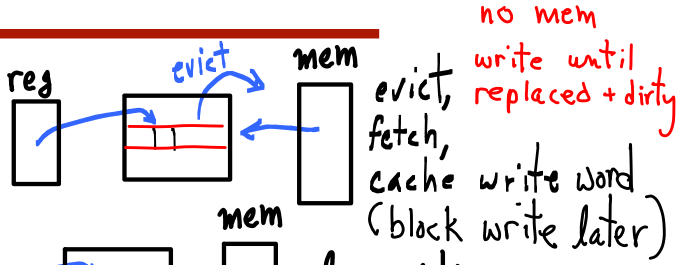
non-dirty write-back, or write-through

Mem-write block, read block to cache, cache-write word

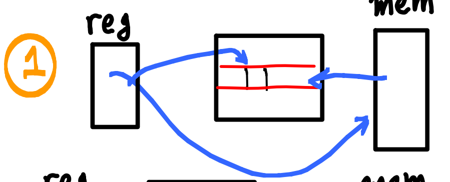
write-back w/ dirty replacement

Write Miss —— Typical Choices

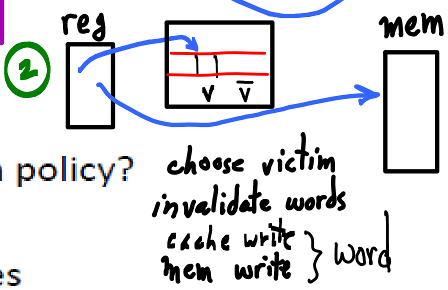
- Write-back caches
  - Write-allocate, fetch-on-miss (why?)
- Write-through caches
  - ① Write-allocate, fetch-on-miss
  - ② Write-allocate, no-fetch-on-miss
  - ③ No-write-allocate, write-around
- Which program patterns match each policy?
- Modern HW support multiple policies
  - Selected by OS on at some coarse granularity (e.g. 4KB)

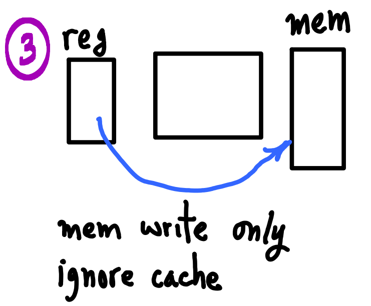


reg
evict
mem
evict, fetch, cache write word (block write later)
no mem write until replaced + dirty



choose victim
fetch
mem/cache write word





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)

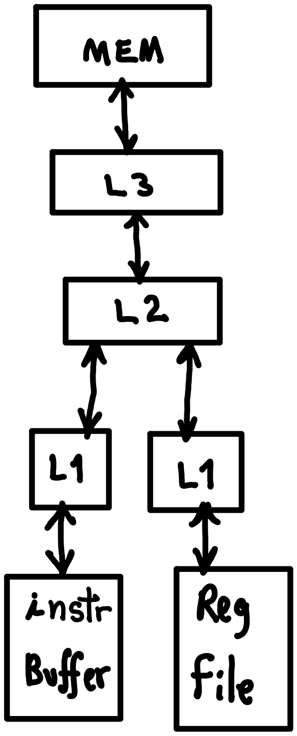
vs.

- 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? (write-through?)

Splitting Caches



- Most processors have separate caches for instructions & data
  - Often noted ad $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 L2

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just a primary (L1) cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

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

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

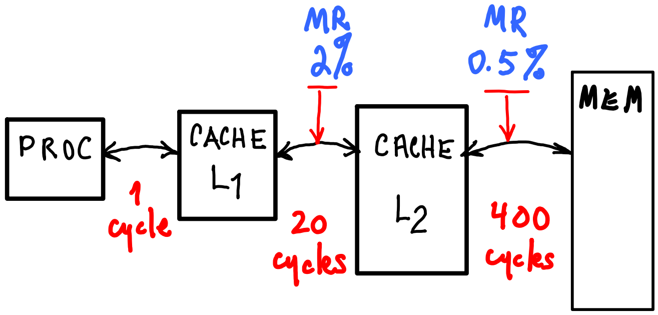
$$\overline{CPI} = (98\%)(1 \text{ cycle for hit}) + \overset{MR}{(2\%)}(400 \text{ cycle stall} + 1 \text{ cycle})$$

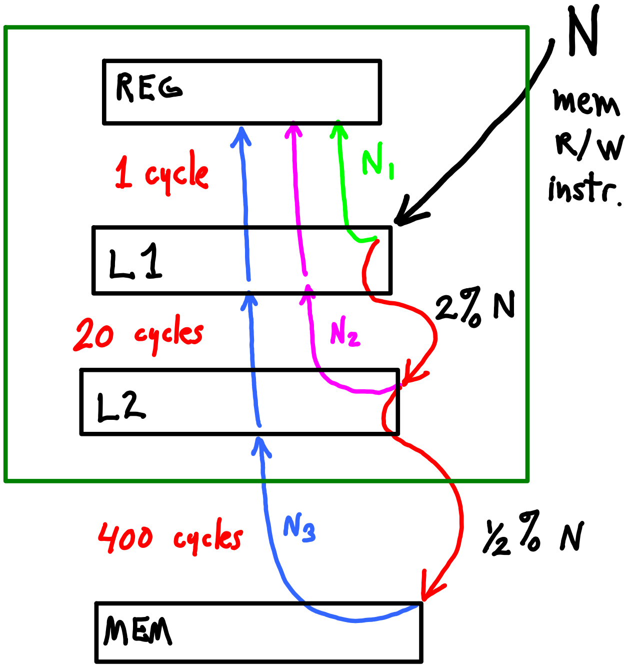
$$= 0.98 + 0.02(400) + 0.02(1) = 1 + 0.02(400) = 9$$

E.G. w/ L2

- 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

C. Kozyrakis





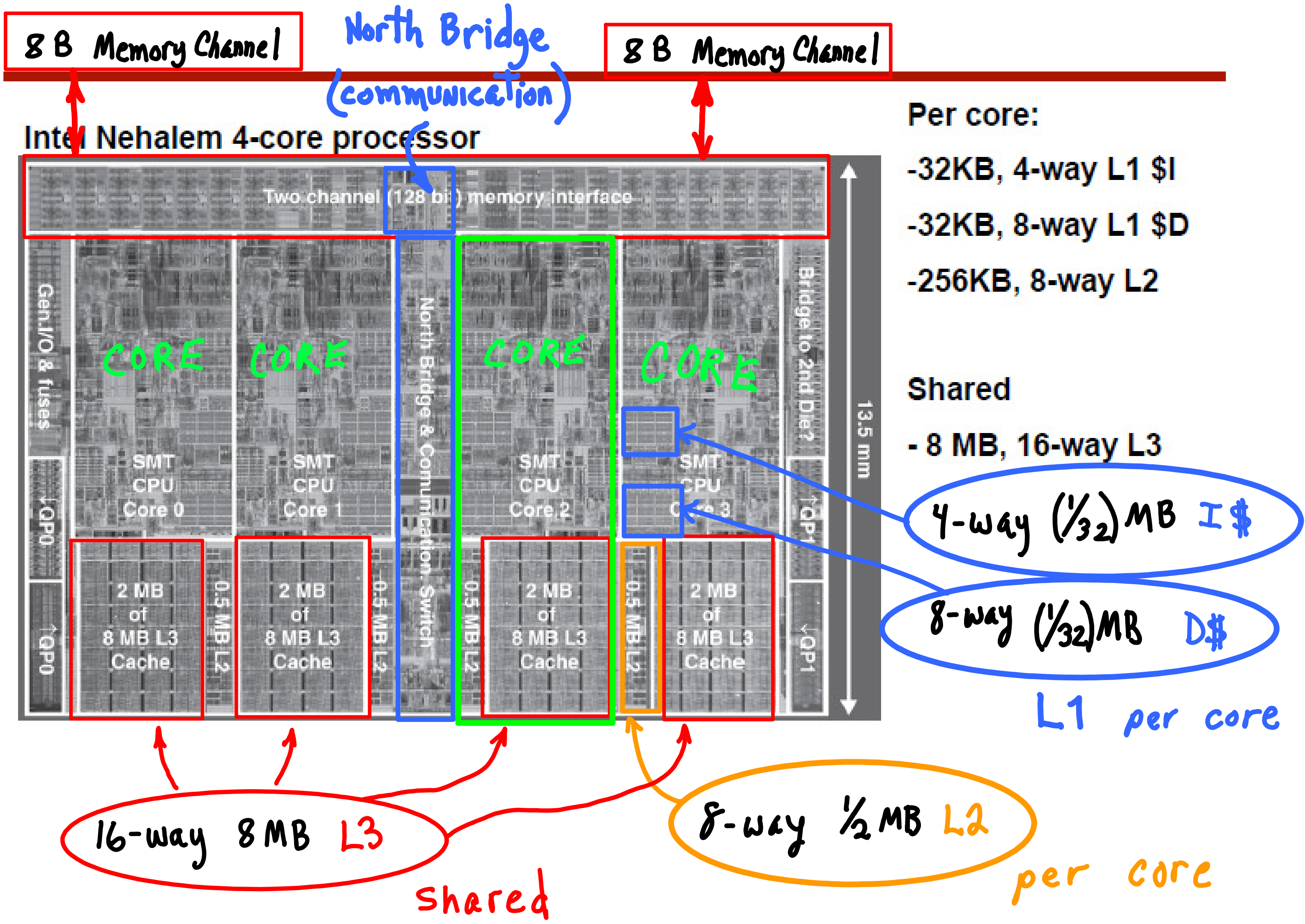
$$\overline{CPI} = \frac{\# \text{cycles}}{N \text{ instructions}}$$

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

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

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

Intel Nehalem 4-core processor



Per core:

-32KB, 4-way L1 $I

-32KB, 8-way L1 $D

-256KB, 8-way L2

Shared

- 8 MB, 16-way L3

all: 64-B blocks, Write-Back Allocate

	Intel Nehalem P6 Quad	AMD Opteron X4
L1 caches (per core) $I $D	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles 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

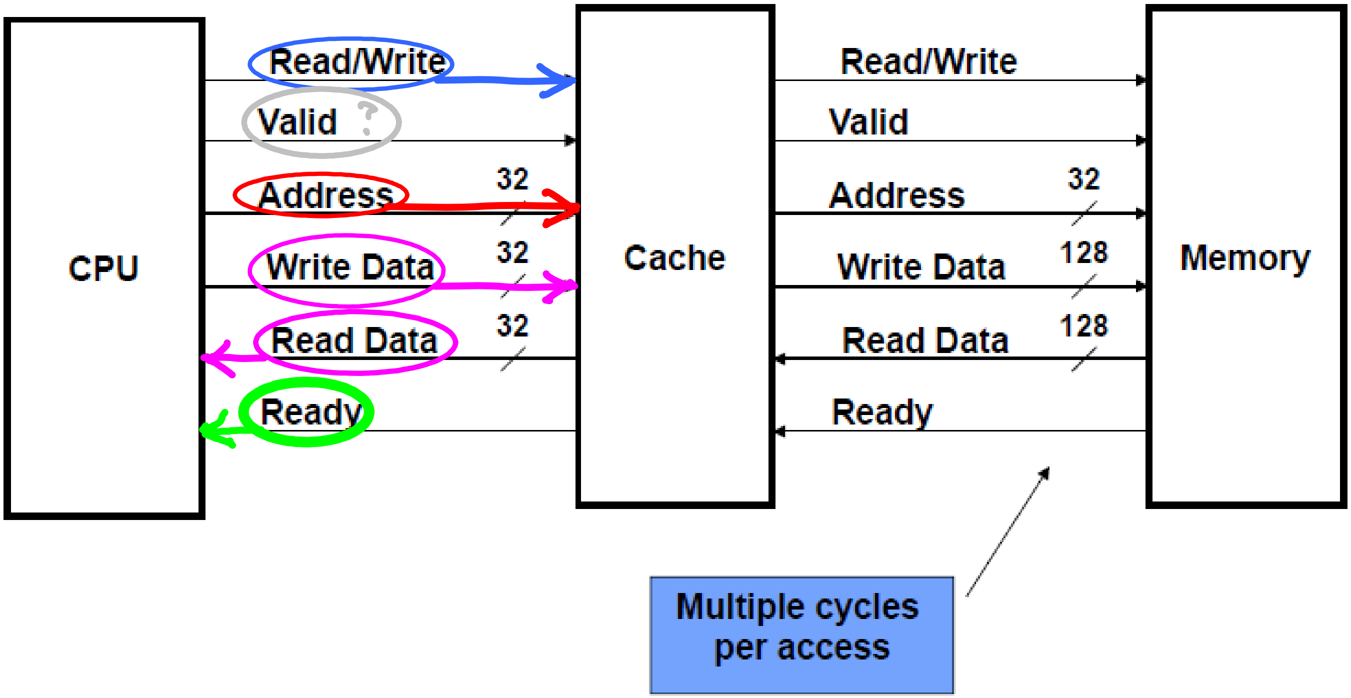
hit 9 cycles

hit 38 cycles

n/a: data not available

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

Interface Signals



CPU

Read/Write

Valid ?

Address 32

Write Data 32

Read Data 32

Ready

Cache

Read/Write

Valid

Address 32

Write Data 128

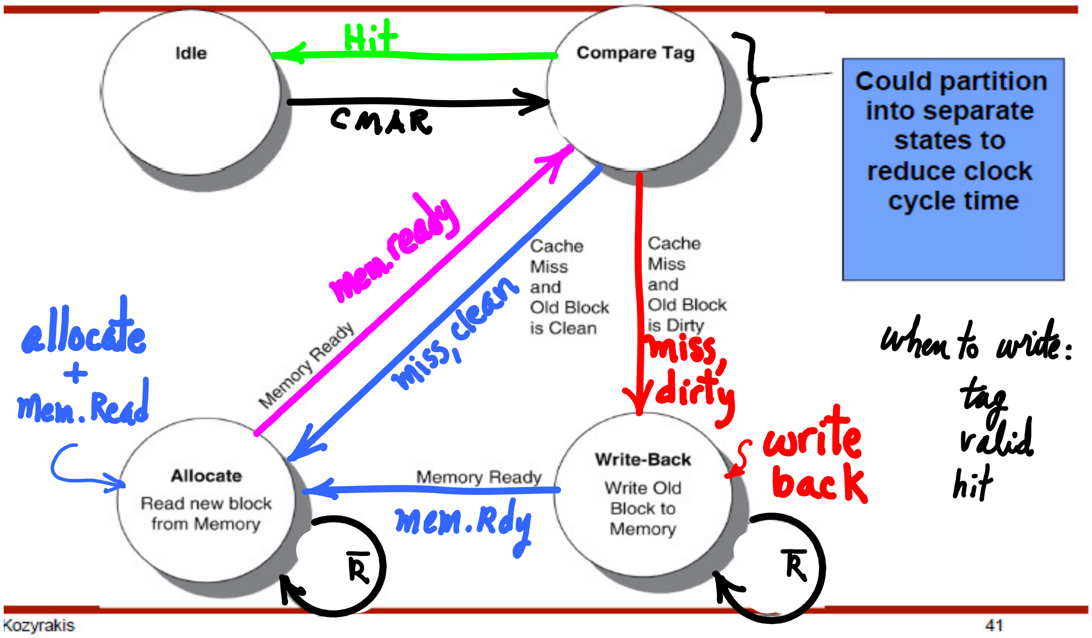
Read Data 128

Ready

Memory

Multiple cycles per access

Cache Controller FSM



Idle

Hit

CMAR

Compare Tag

Could partition into separate states to reduce clock cycle time

mem.ready

Memory Ready

miss, clean

Cache Miss and Old Block is Clean

Cache Miss and Old Block is Dirty

miss, dirty

allocate + mem.Read

Allocate

Read new block from Memory

$\overline{R}$

Memory Ready

mem.Rdy

Write-Back

Write Old Block to Memory

write back

$\overline{R}$

when to write:

tag

valid

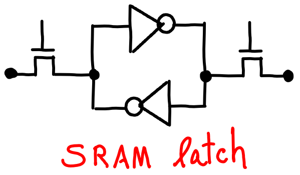
hit

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

Memory Technologies

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

a latch

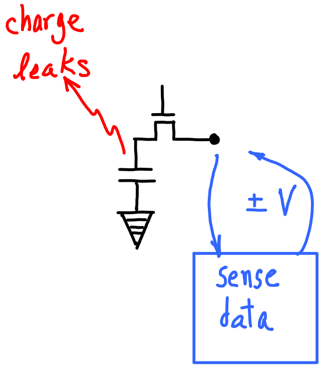


- DRAM
  - Must be re-written after being read
  - Must also be periodically refeshed
    - Every ~ 8 ms
    - Each row can be refreshed simultaneously
  - One transistor/bit
  - Address lines are multiplexed:
    - Upper half of address: row access strobe (RAS)
    - Lower half of address: column access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

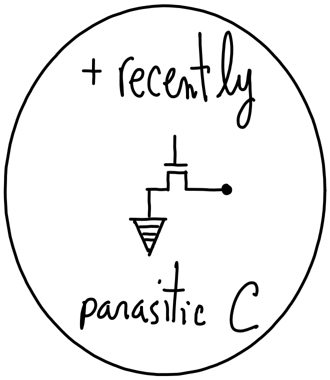
refresh

a capacitor + switch

½ size addr bus



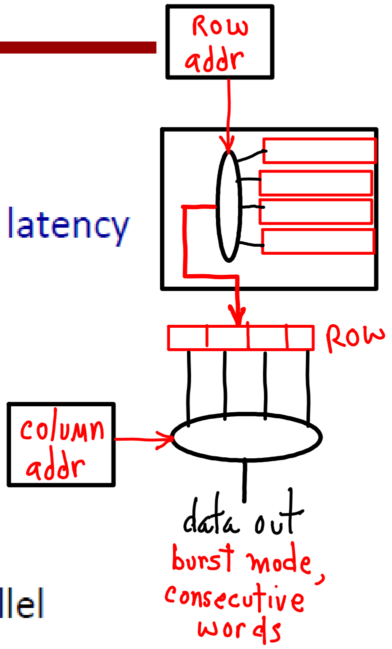
- Some optimizations:
  - Multiple accesses to same row
  - Synchronous DRAM
    - Added clock to DRAM interface
    - Burst mode with critical word first
  - Wider interfaces
  - Double data rate (DDR)
  - Multiple banks on each DRAM device



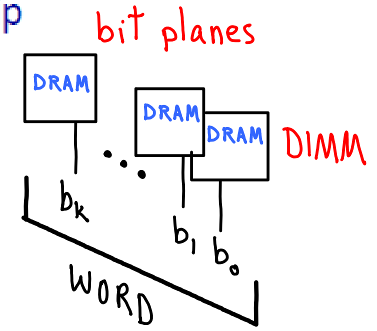
Transfer on rising + falling edges

DRAM

- 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 × 2 clocks
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM DDR × 2 data bus (in, out)
  - Four transfers per cycle



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

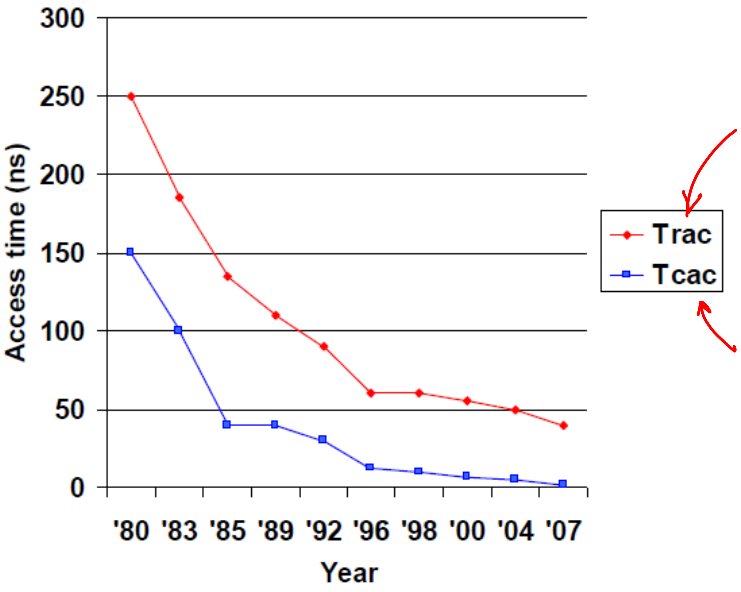


			Row access strobe (RAS)			
Production year	Chip size	DRAM Type	Slowest DRAM (ns)	Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	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
	$\times 2^{17} = 1/8$ M				× 150	× 9

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

× $3 \cdot 10^4$

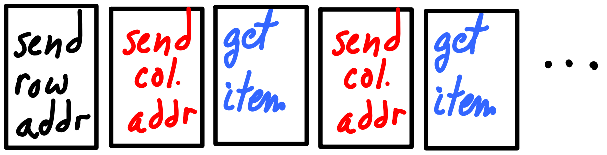


Total Time for Random address

delay per column

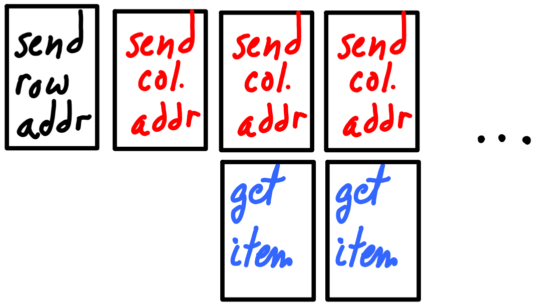
Improving DRAM bandwidth (other than faster cycle time)

Fast Page Mode



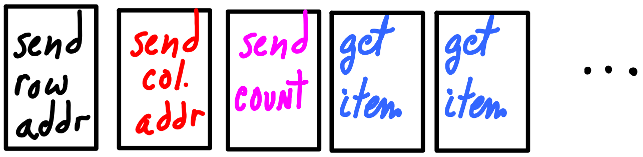
Access multiple items in same row.

Etended Data Out

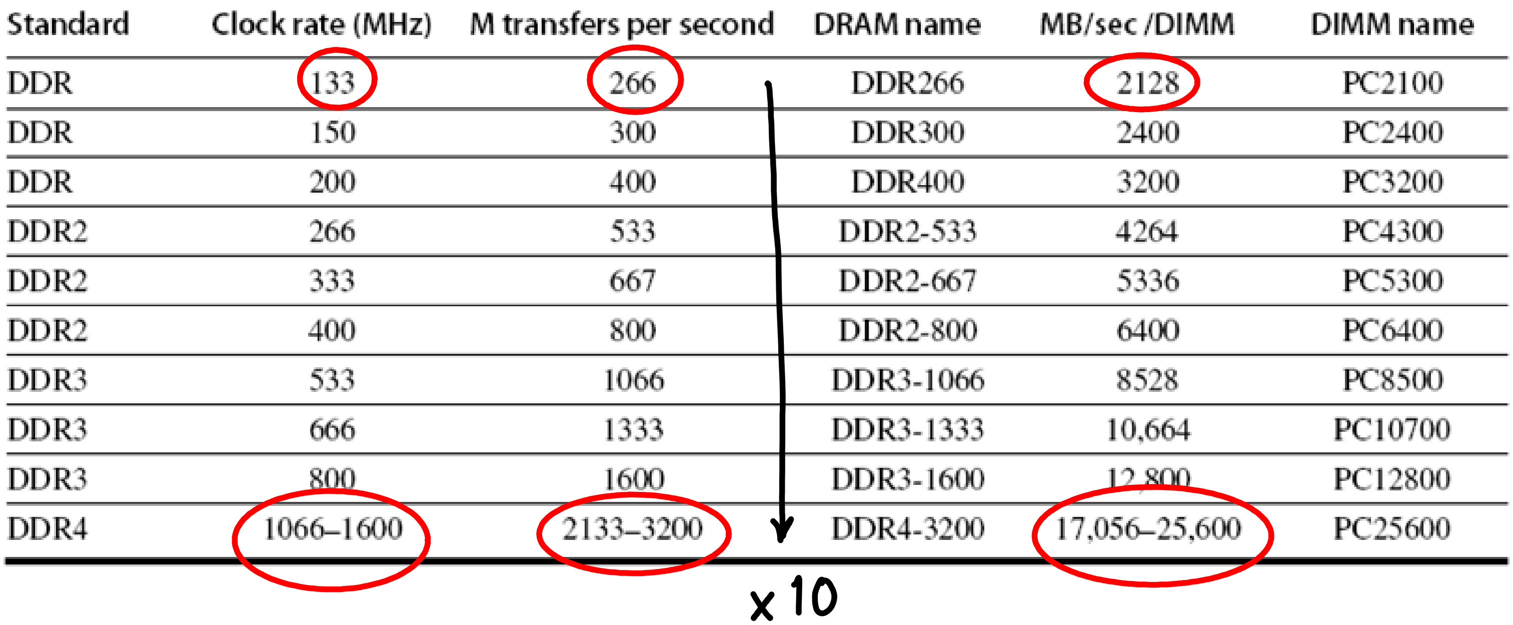


Overlap sending column address with
Accessing item

BURST EDO



Only send column address once, then
send count of items to access

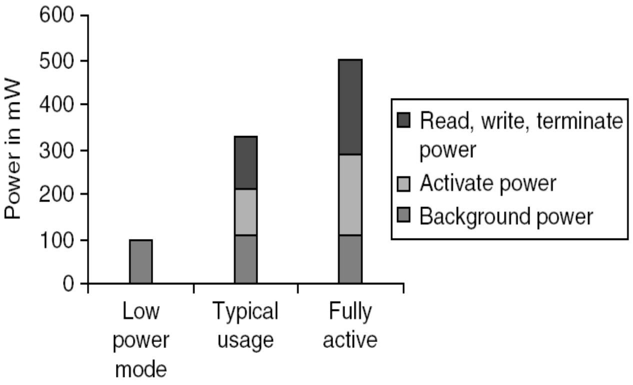


Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	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

x 10

- 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

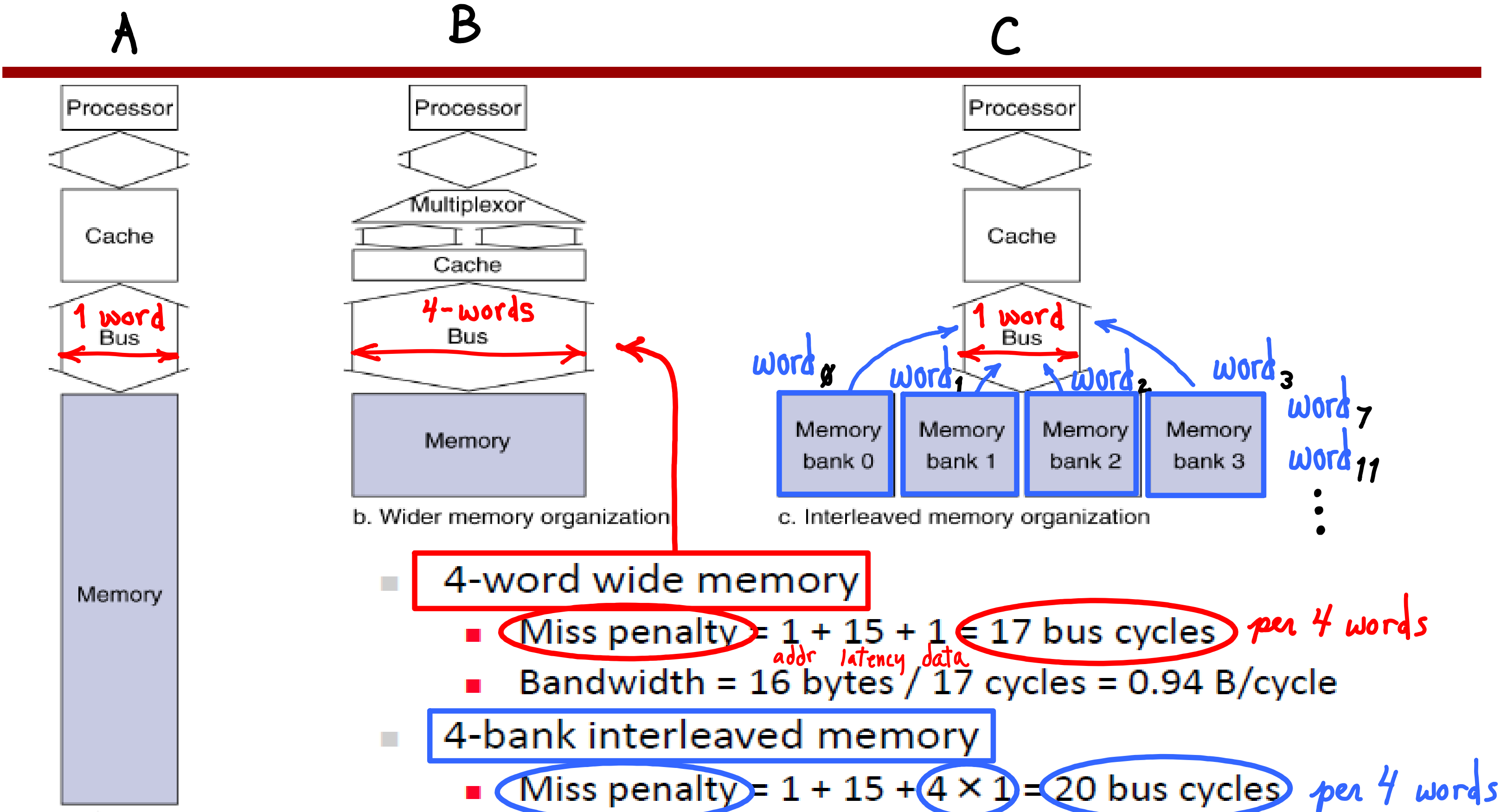
- Graphics memory:
  - Achieve 2-5 X bandwidth per DRAM vs. DDR3
    - Wider interfaces (32 vs. 16 bit)
    - Higher clock rate
      - Possible because they are attached via soldering instead of socketted DIMM modules



- 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



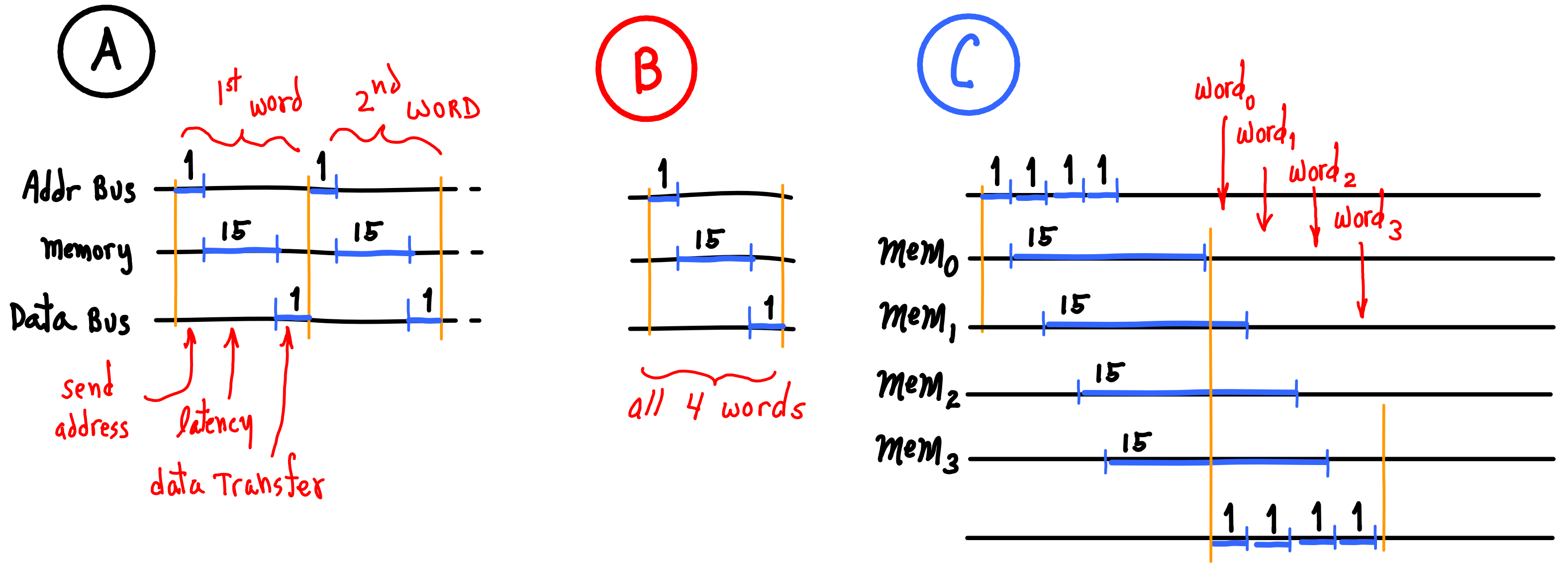
a. One-word-wide memory organization

b. Wider memory organization

c. Interleaved memory organization

- 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles per 4 words
    addr latency data
  - 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

Bus Cycle Timing, 4-word Access



$(1+15+1) \times 4 / 4$ words

$(1+15+1) / 4$ words

$1+15+(4\times1) / 4$ words

- Six basic cache optimizations:
  - Larger block size
    - 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
  - Avoiding address translation in cache indexing
    - Reduces hit time