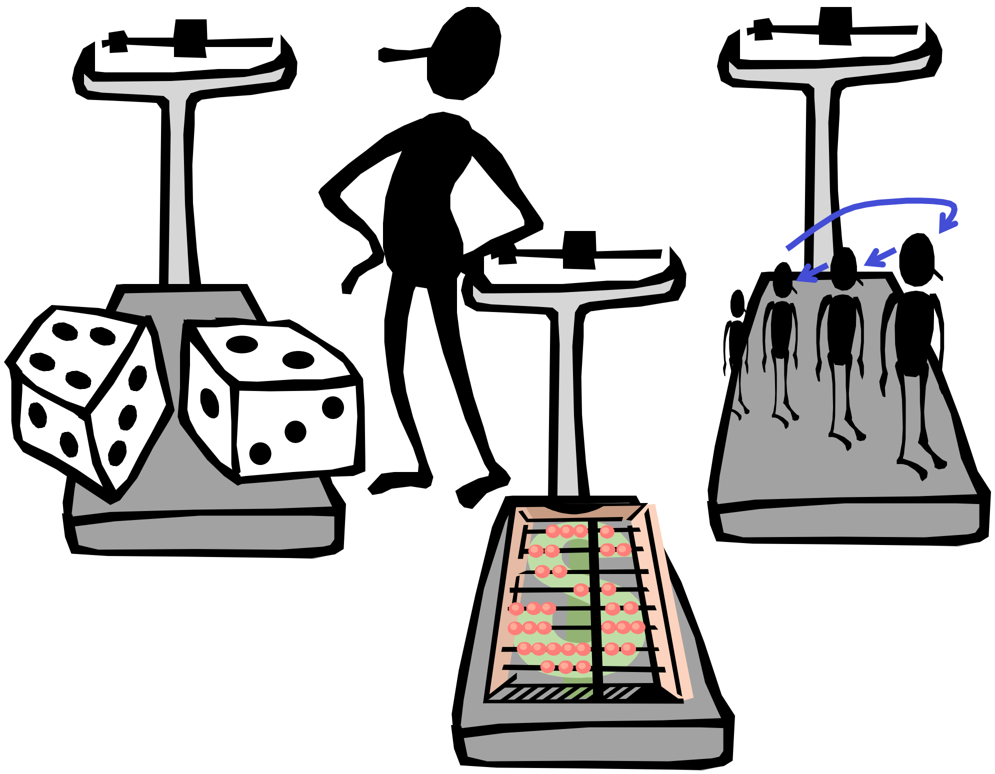
Cache Structure

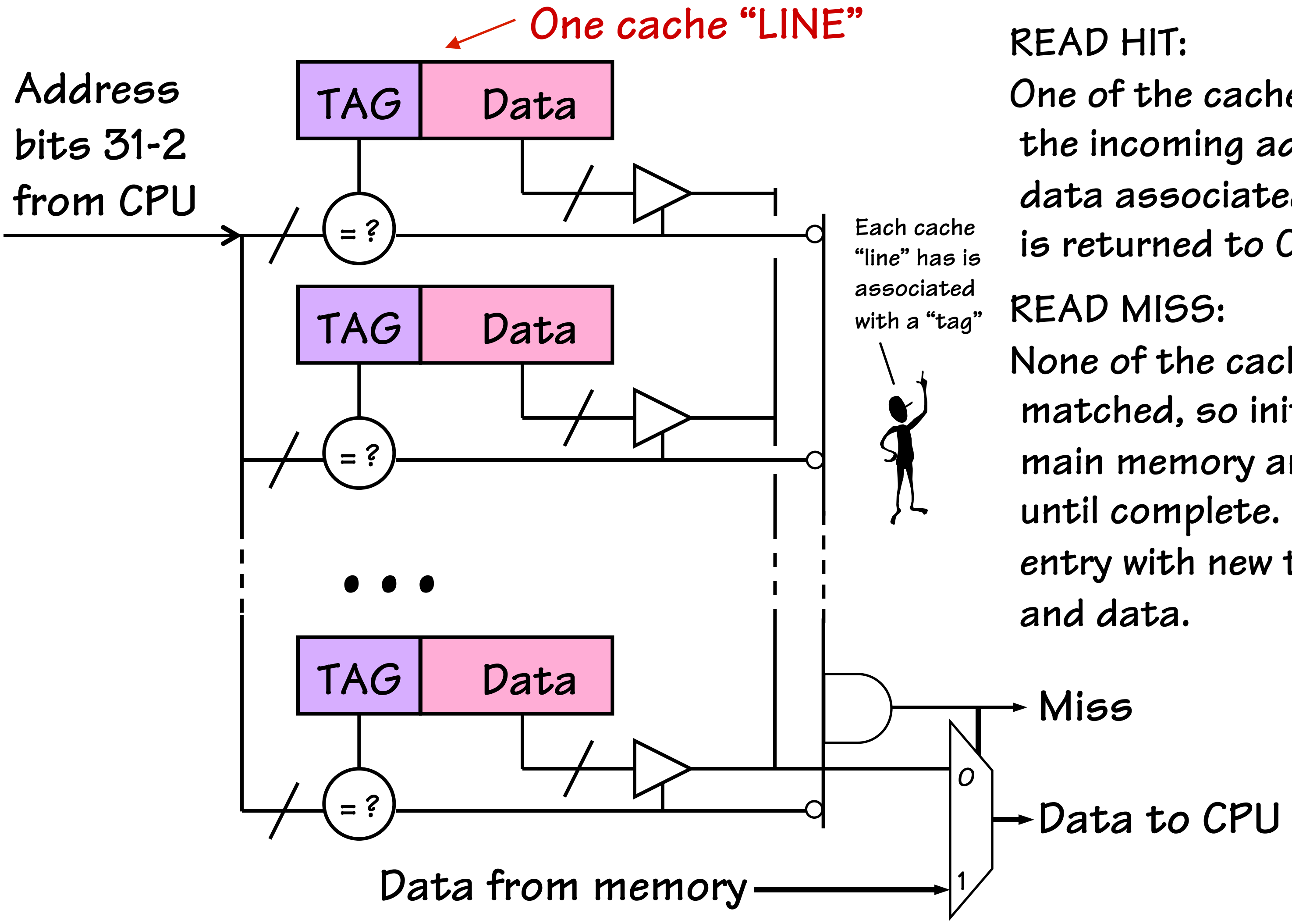


- Set-Associativity
- Replacement policies
  - Overhead
  - Implementation
- Handling writes
- Cache simulations

Study 5.3, 5.5

Fully Associative Cache

(from last Lecture)



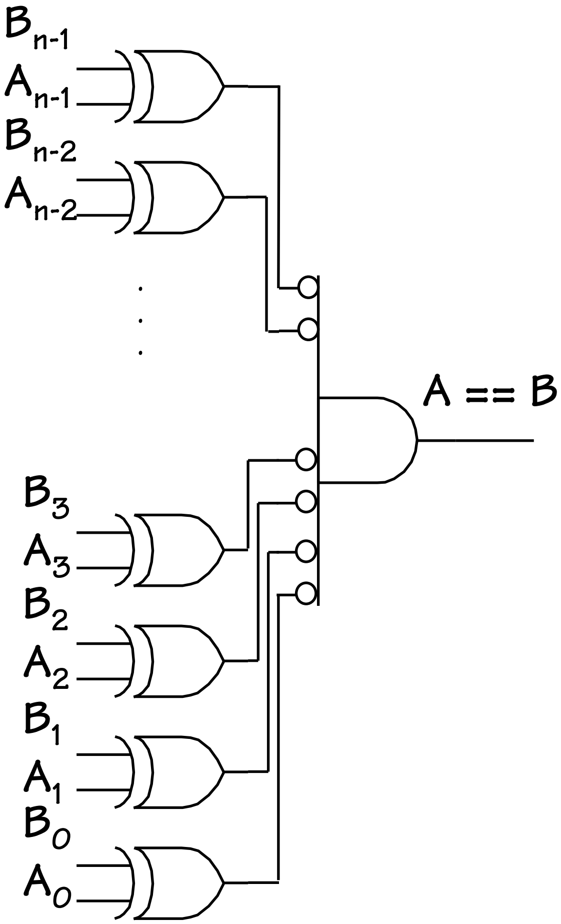
READ HIT:

One of the cache tags matches the incoming address; the data associated with that tag is returned to CPU.

READ MISS:

None of the cache tags matched, so initiate access to main memory and stall CPU until complete. Update cache entry with new tag (address) and data.

Tags Are Expensive!



- Tag <u>comparison logic</u> is LARGE

  An XOR gate is as large as a memory bit!

- Tag <u>comparison logic</u> is SLOW

  High-Fan-In NOR gate

- Tag <u>storage overhead</u> is high

  Rather store data, not Tags

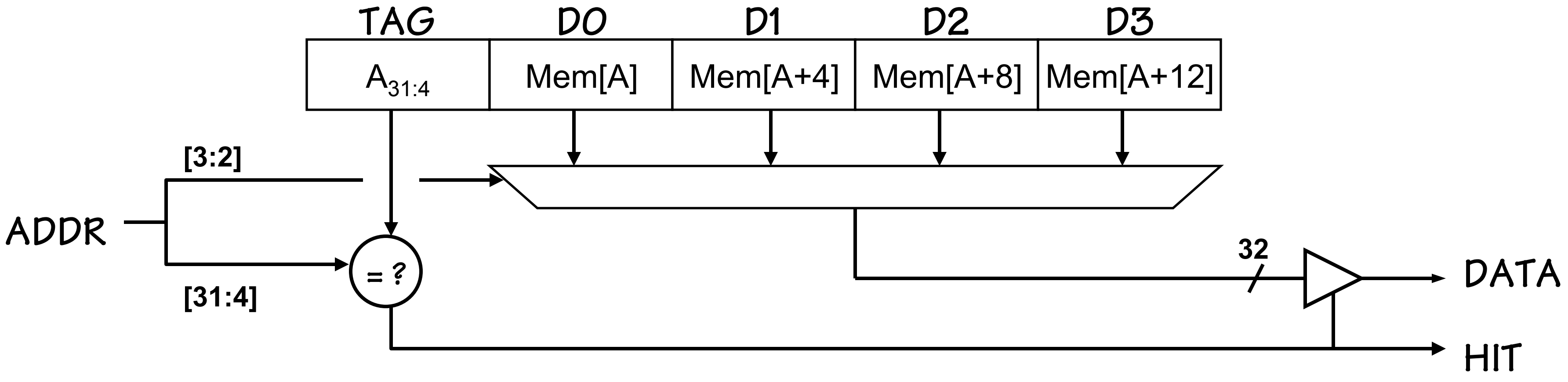
e.g. For a Fully Associative Cache

Tag bits/cache entry = 30 bits
Data bits/cache entry = 32 bits

48% of cache's memory is devoted to tag storage!

Amortize Tag Costs: More Data/Tag

BIG Cache Lines: Enlarge each line in fully-associative cache



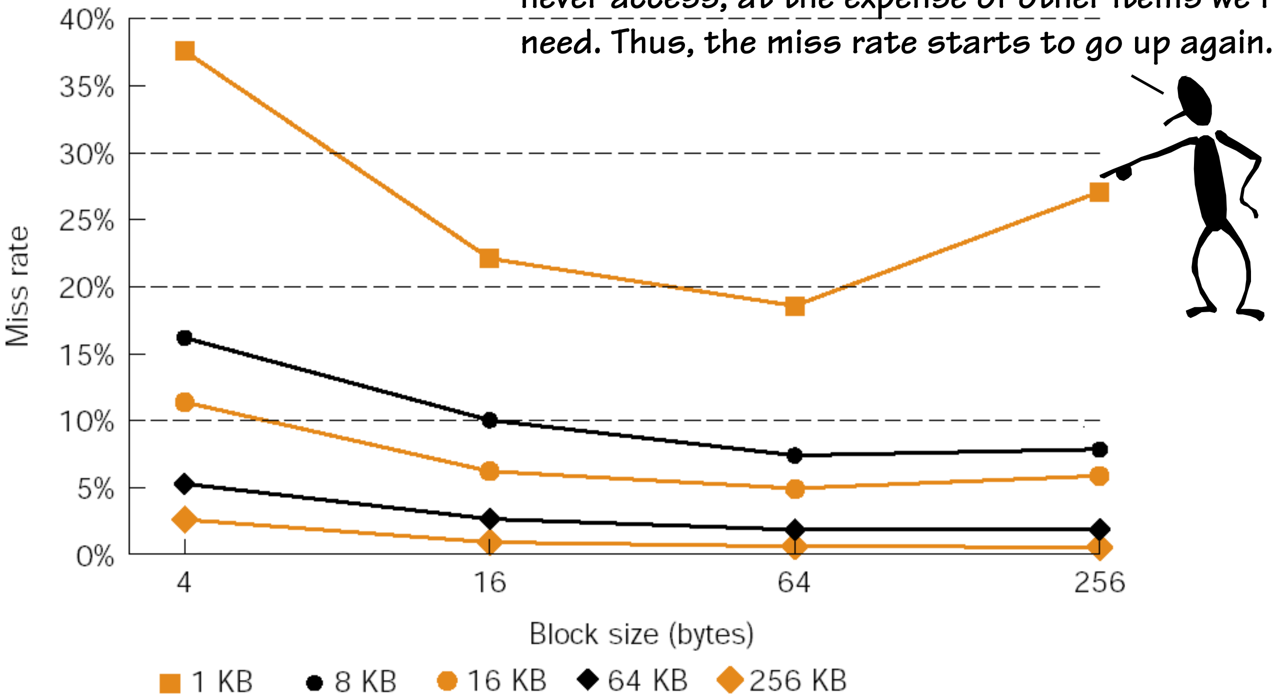
- Blocks of 2^B words, on 2^B word boundaries
- always reads/writes a 2^B word BLOCK from/to memory
- exploits spatial locality: nearby words in block, likely to accessed
- cost: some fetches of unaccessed words
- BIG WIN if path to memory is wide, or sequential accesses are fast

Tag bits/cache entry = (30 – 2) bits

Data bits/cache entry = 4*32 bits   Only 18% of cache's memory used for tags

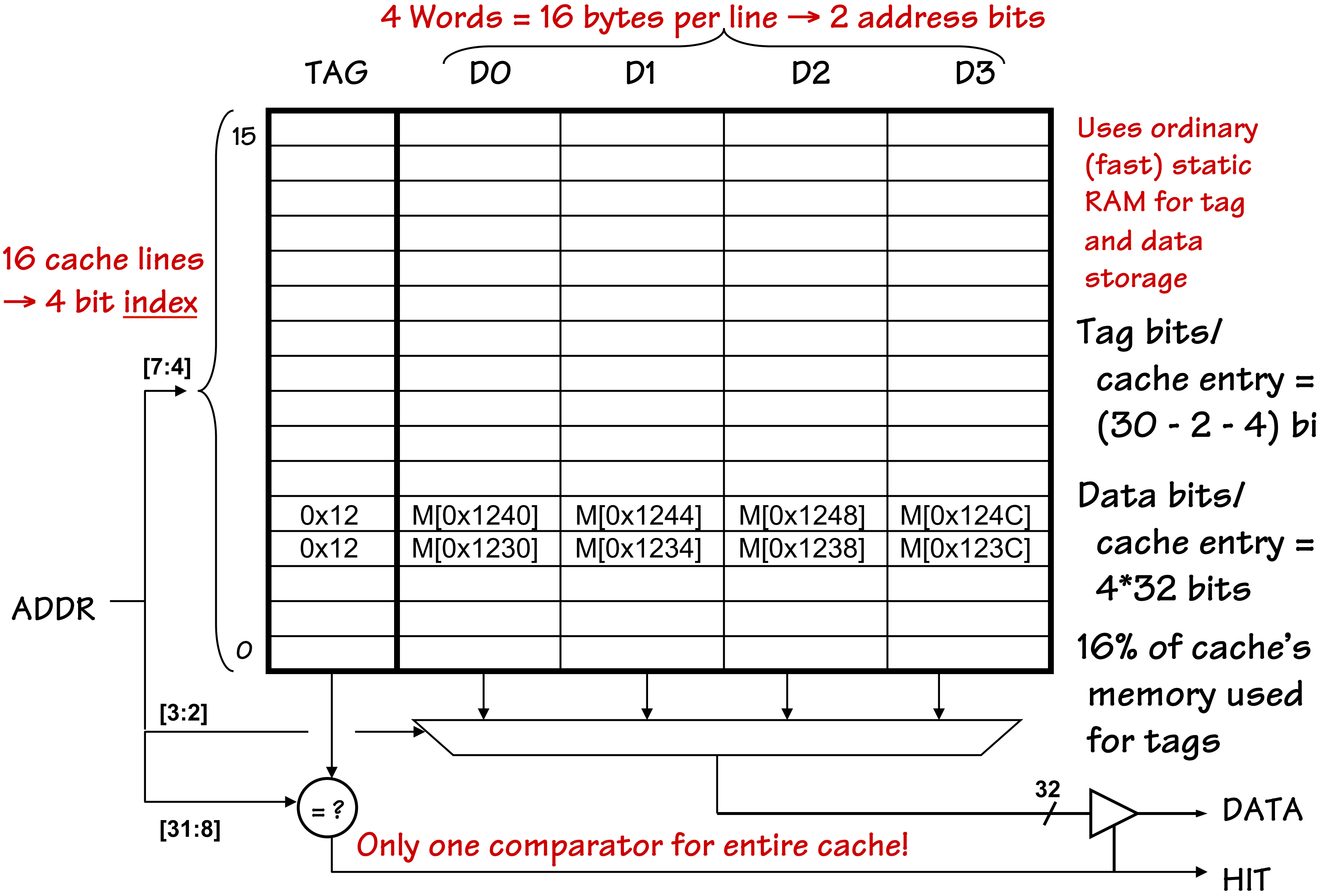
Block Size vs. Miss Rate

If the block size gets too big, we start fetching items that we never access, at the expense of other items we might actually need. Thus, the miss rate starts to go up again.



- spatial locality: larger blocks → reduce miss rate
- fixed cache size: larger blocks
  - → fewer lines in cache
  - → higher miss rate, especially in small caches

Direct-Mapped Cache (from last time)



4 Words = 16 bytes per line → 2 address bits

16 cache lines
→ 4 bit index

Only one comparator for entire cache!

Uses ordinary (fast) static RAM for tag and data storage

Tag bits/ cache entry = (30 - 2 - 4) bits

Data bits/ cache entry = 4*32 bits

16% of cache's memory used for tags

Fully-Assoc. vs. Direct-mapped

Fully-associative N-line cache:

- N tag comparators, registers used for tag/data storage ($$$)
- Location A can be stored in ANY of the N cache lines; no "collisions"
- Replacement strategy (e.g., LRU) used to pick which line to use when loading new word(s) into cache

COLLISIONs occur when there multiple items that we'd like to keep cached, we have room, but our management policies allows us to only a subset of them.

Direct-mapped N-line cache:

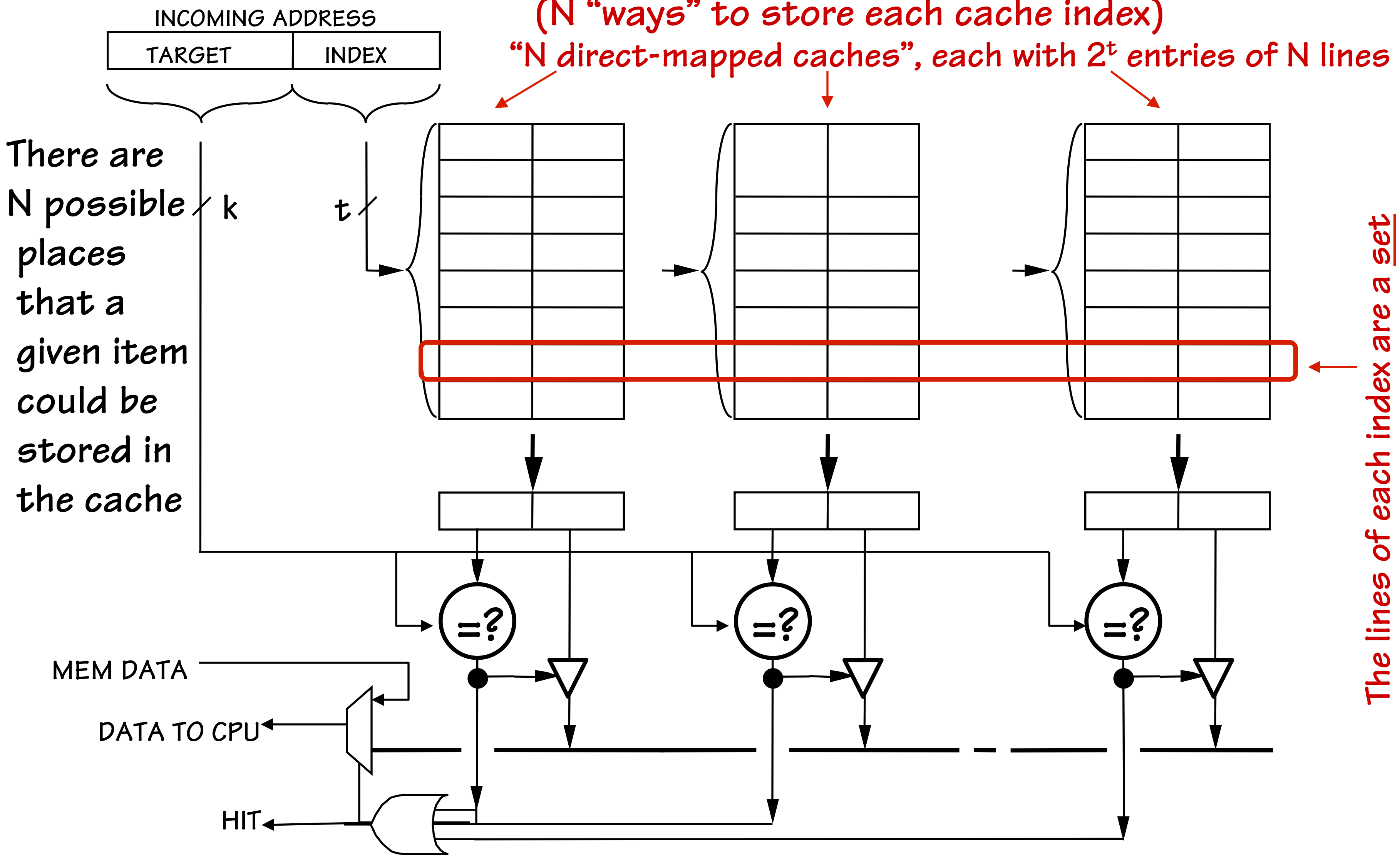
- 1 tag comparator, SRAM used for tag/data storage ($)
- Location A is stored in a SPECIFIC line of the cache determined by its address; address "collisions" possible
- Replacement strategy not needed: each word can only be cached in one specific cache line

Is there something in-between?

N-Way Set-Associative Cache

(N "ways" to store each cache index)

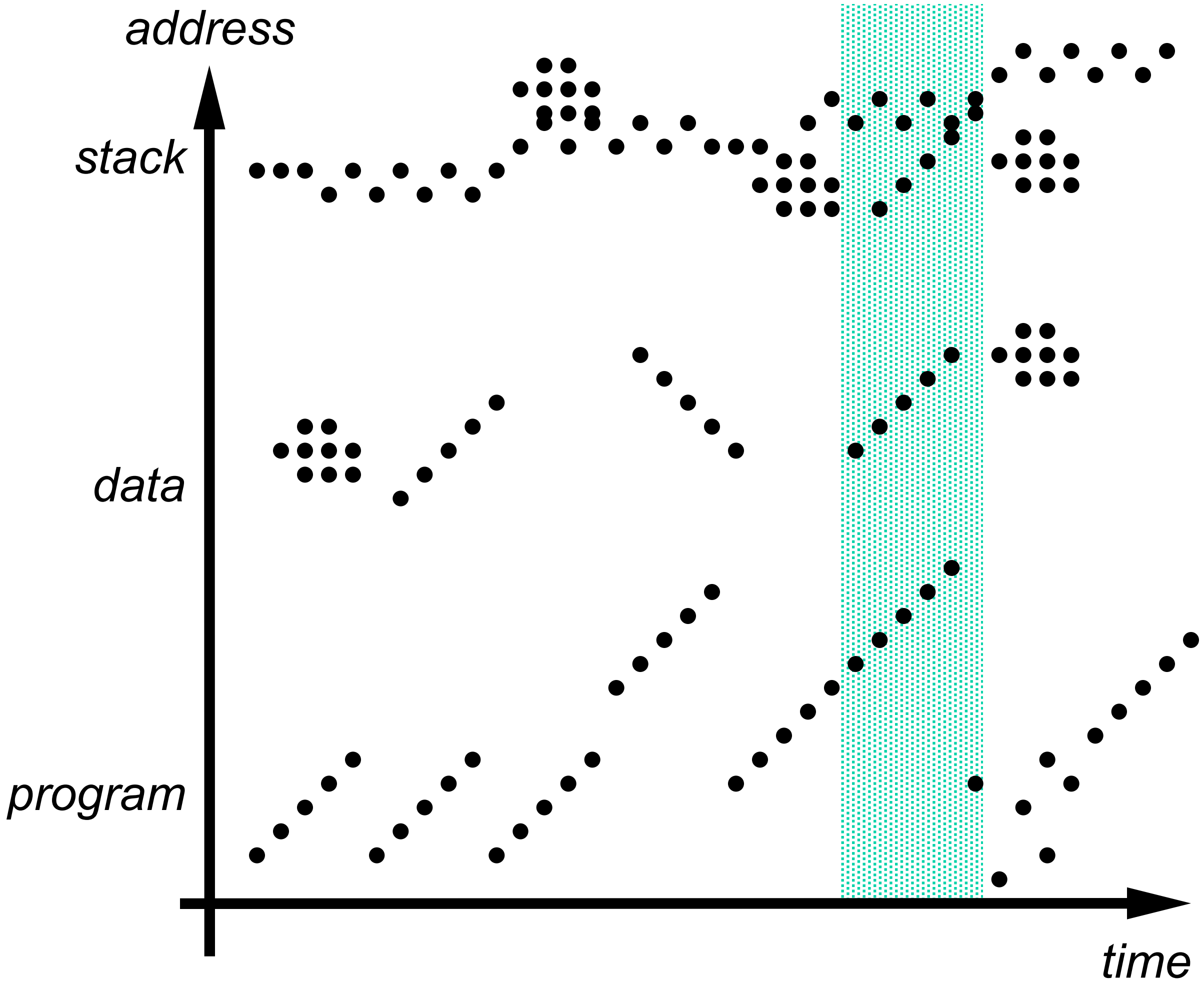
"N direct-mapped caches", each with 2^t entries of N lines



There are N possible places that a given item could be stored in the cache

The lines of each index are a <u>set</u>

How Many Lines in a Set?

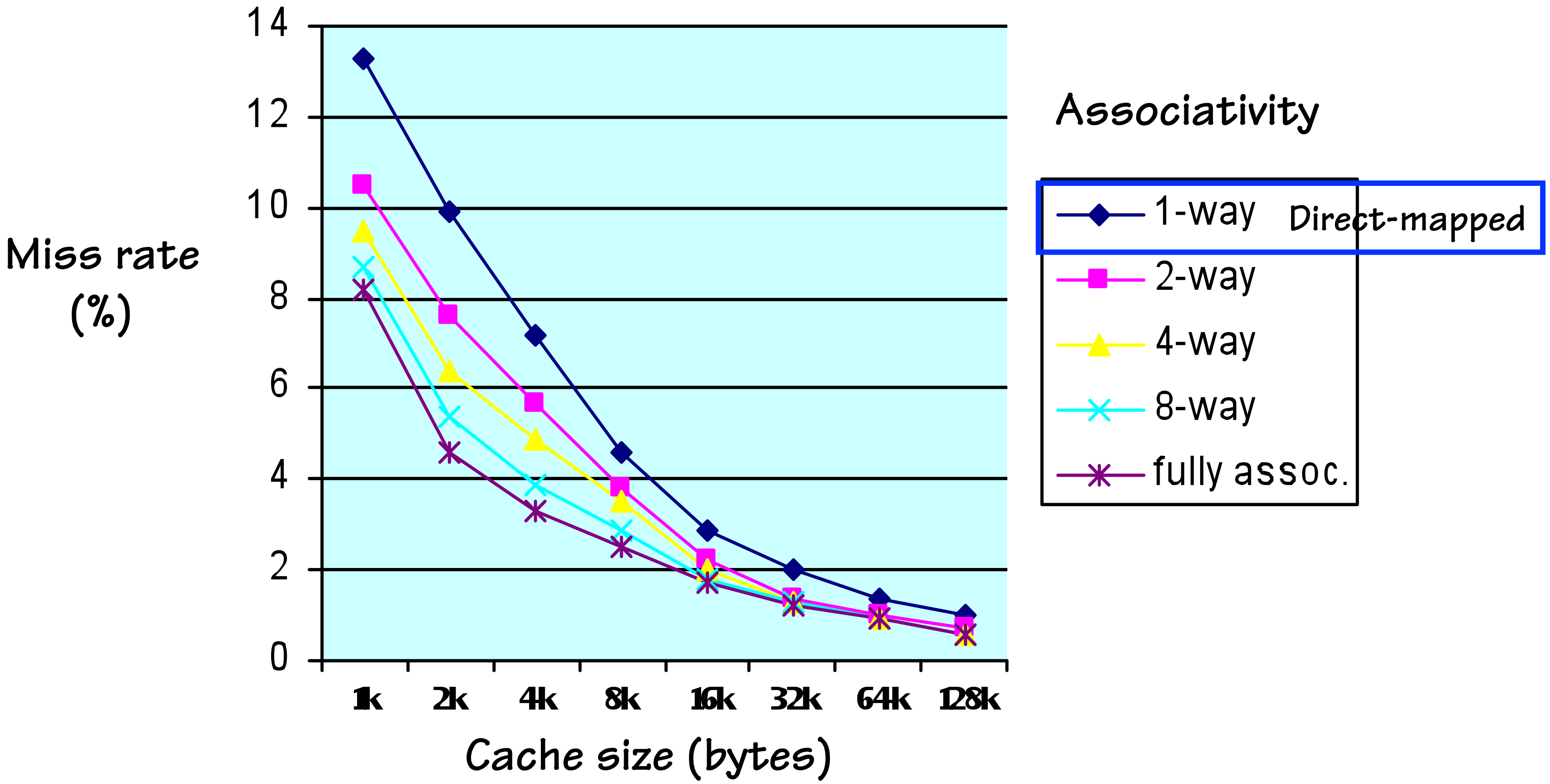


② stack

③ ④ data (src, dest)

① instructions

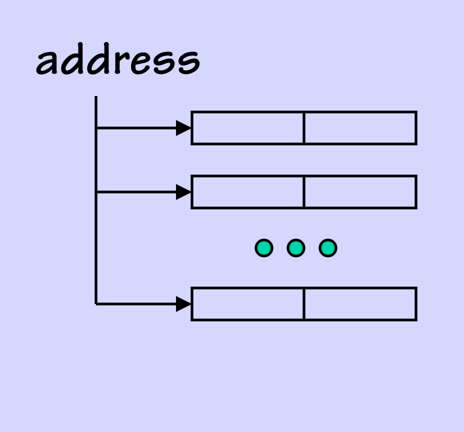
Associativity vs. Miss Rate



- 8-way is (almost) as effective as fully-associative
- rule of thumb: N-byte M-way set assoc ≈ N/2-byte 2M-way set assoc.

Continuum of Associativity

Fully associative

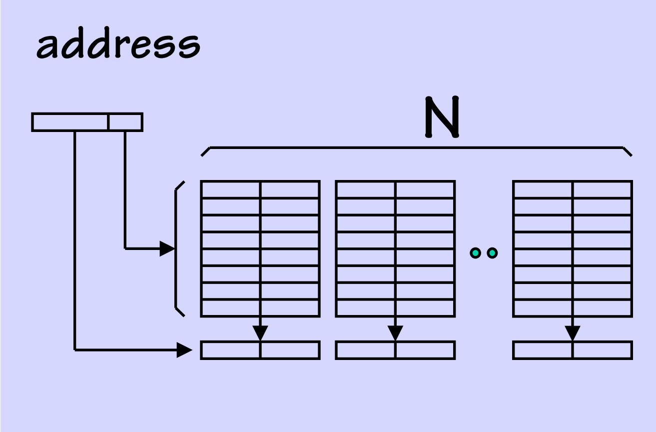


- compares addr with all tags simultaneously
- location A can be stored in any cache line

ON A MISS?

Allocates a cache entry

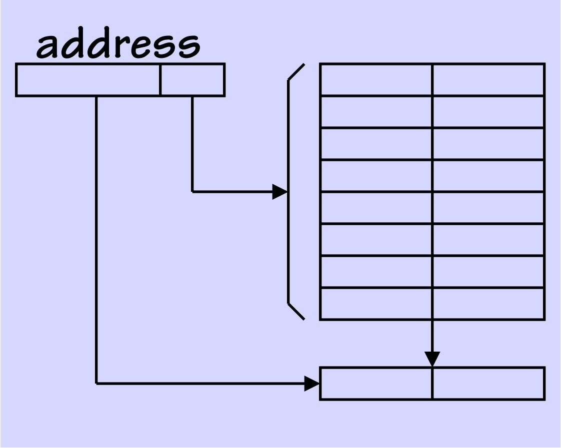
N-way set associative



- compares addr with N tags simultaneously
- Data can be stored in any of the N cache lines belonging to a "set"
- like N Direct-mapped caches

Allocates 1 of N lines in a set

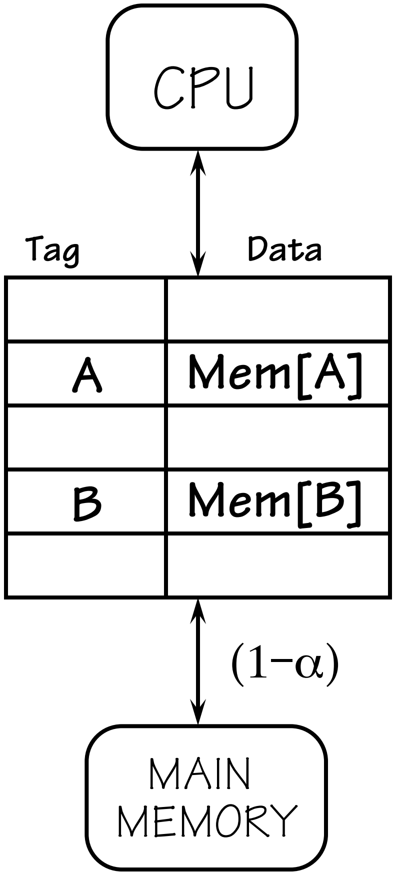
Direct-mapped



- compare addr with only one tag
- location A can be stored in exactly one cache line

Only one place to put it

Basic Caching Algorithm



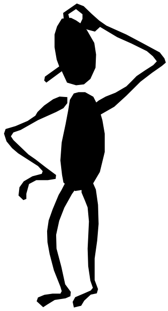
ON REFERENCE TO Mem[X]: Look for X among cache tags...

HIT: *X = TAG(i) , for some cache line i*

READ: return DATA(i)

WRITE: change DATA(i);
Write to Mem[X]

MISS: *X not found in TAG of any cache line*



Our next focus

REPLACEMENT ALGORITHM:
Select some LINE k to hold Mem[X] (Allocation)

READ: Read Mem[X]
Set TAG(k)=X, DATA(k)=Mem[X]

WRITE: Write to Mem[X]
Set TAG(k)=X, DATA(k)= write data

Three Replacement Strategies

LRU (Least-recently used)

- replaces the item that has gone UNACCESSED the LONGEST
- favors the most recently accessed data

FIFO/LRR (first-in, first-out/least-recently replaced)

- replaces the OLDEST item in cache
- favors recently loaded items over older STALE items

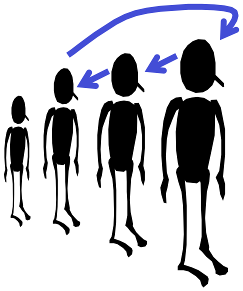
Random

- replace some item at RANDOM
- no favoritism – uniform distribution
- no "pathological" reference streams causing worst-case results
- use pseudo-random generator to get reproducible behavior

Keeping Track of LRU

- Needs to keep ordered list of N items for an N-way associative cache, that is *updated on every access*. Example for N = 4:

Current Order	Action	Resulting Order
(0,1,2,3)	Hit 2	(2,0,1,3)
(2,0,1,3)	Hit 1	(1,2,0,3)
(1,2,0,3)	Miss, Replace 3	(3,1,2,0)
(3,1,2,0)	Hit 3	(3,1,2,0)



- N! possible orderings → $\log_2 N!$ bits per set approx $O(N \log_2 N)$ "LRU bits" + update logic

Example: LRU for 2-Way Sets

- Bits needed? $\log_2 2! = 1$ per set
- LRU bit is selected using the same index as cache (Part of same SRAM)
- Bit keeps track of the last line accessed in set:

  (0), Hit 0 -> (0)

  (0), Hit 1 -> (1)

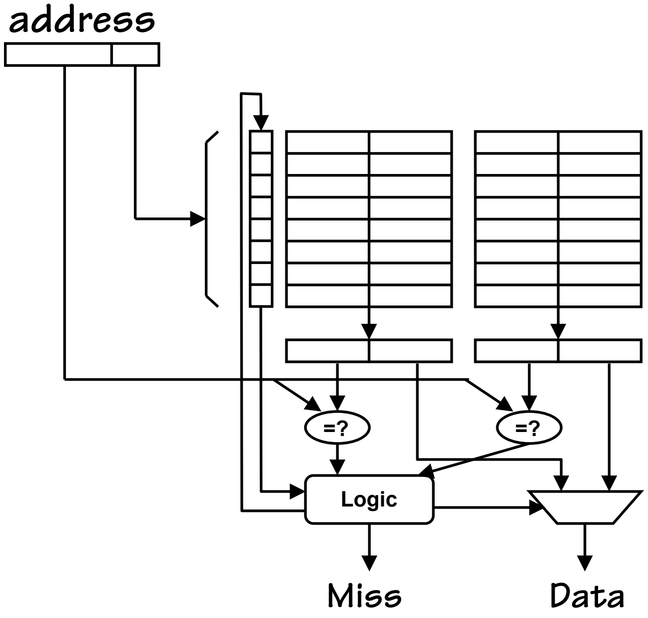
  (0), Miss, replace 1 -> (1)

  (1), Hit 0 -> (0)

  (1), Hit 1 -> (1)

  (1), Miss, replace 0 -> (0)

2-way set associative



Example: LRU for 4-Way Sets

- Bits needed? $\log_2 4! = \log_2 24 = 5$ per set
- How?
- One Method:
  "One-Out/Hidden Line" coding (and variants)

Directly encode the indices of the N-2 most recently accessed lines, plus one bit indicating if the smaller (0) or larger (1) of the remaining lines was most recently accessed

(2,0,1,3) -> 10 00 0
(3,2,1,0) -> 11 10 1
(3,2,0,1) -> 11 10 0

Requires $(N-2)*\log_2 N + 1$ bits

- 8-Way sets? $\log_2 8! = 16$, $(8-2)*\log_2 8 + 1 = 19$

Bottom line, LRU replacement requires considerable overhead as associativity increases

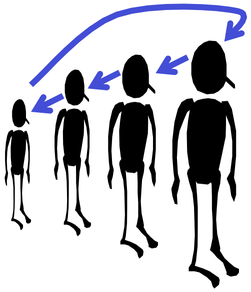
Overhead is $O(N \log_2 N)$ bits/set

FIFO Replacement

- Each set keeps a modulo-N counter that points to victim line that will be replaced on the next miss
- Counter is only updated only on cache misses

Ex: for a 4-way set associative cache:

Next Victim	Action
(0)	Miss, Replace 0
(1)	Hit 1
(1)	Miss, Replace 1
(2)	Miss, Replace 2
(3)	Miss, Replace 3
(0)	Miss, Replace 0



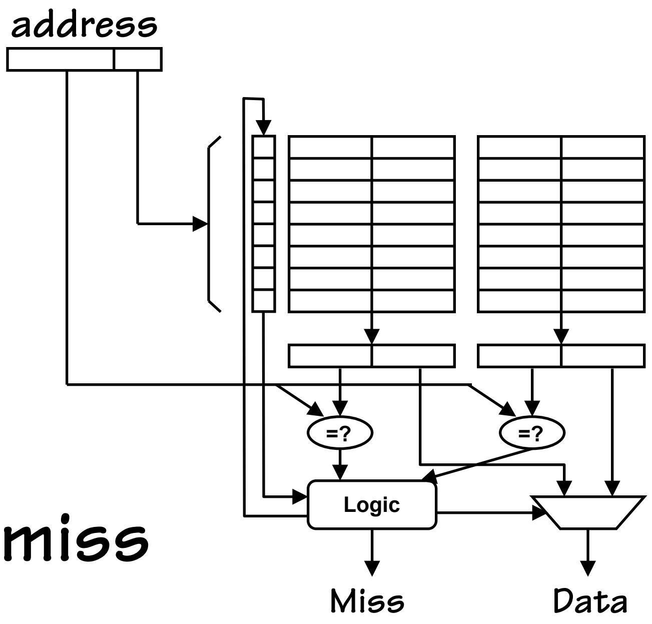
Overhead is $O(\log_2 N)$ bits/set

Example: FIFO For 2-Way Sets

- Bits needed? $\log_2 2 = 1$ per set
- FIFO bit is per cache line and uses the same index as cache
  (Part of same SRAM)
- Bit keeps track of the oldest line in set
- <u>Same overhead as LRU!</u>
- LRU is generally has lower miss rates than FIFO, soooo….

  WHY BOTHER???

2-way set associative



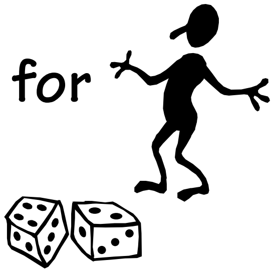
FIFO For 4-way Sets

- Bits Needed? $\log_2 4 = 2$ per set
- Low-cost, easy to implement (no tricks here)
- 8-way? $\log_2 8 = 3$ per set
- 16-way? $\log_2 16 = 4$ per set
- LRU 16-way?

  $\log_2 16! = 45$ bits per set

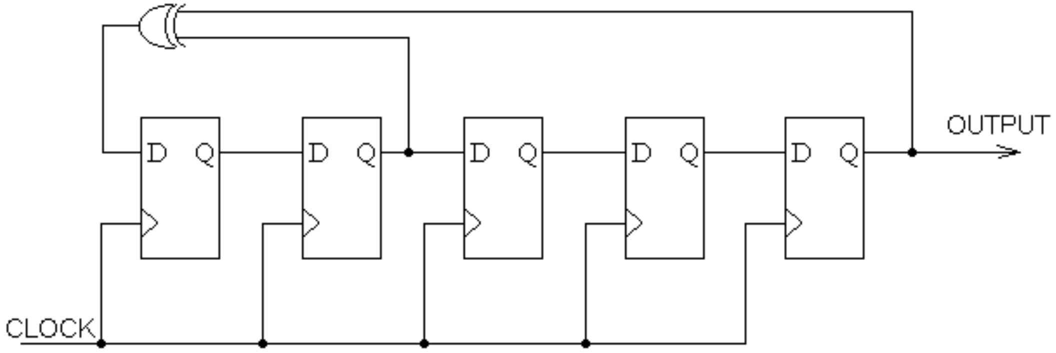
  $14 * \log_2 16 + 1 = 57$ bits per set
- FIFO summary
  - Easy to implement, scales well,
    BUT CAN WE AFFORD IT?

Random Replacement



- Build a single Pseudorandom Number generator for the WHOLE cache. On a miss, roll the dice and throw out a cache line at random.
- Updates only on misses.
- How do you build a random number generator (easier than you might think).

Overhead is $O(\log_2 N)$ bits/cache!



Pseudorandom Linear Feedback Shift Register

Counting Sequence

11111	0x1F	01000	0x08
01111	0x0F	10100	0x14
00111	0x07	01010	0x0A
10011	0x13	10101	0x15
11001	0x19	11010	0x1A
01100	0x0C	11101	0x1D
10110	0x16	01110	0x0E
01011	0x0B	10111	0x17
00101	0x05	11011	0x1B
10010	0x12	01101	0x0D
01001	0x09	00110	0x06
00100	0x04	00011	0x03
00010	0x02	10001	0x11
00001	0x01	11000	0x18
10000	0x10	11100	0x1C
		11110	0x1E

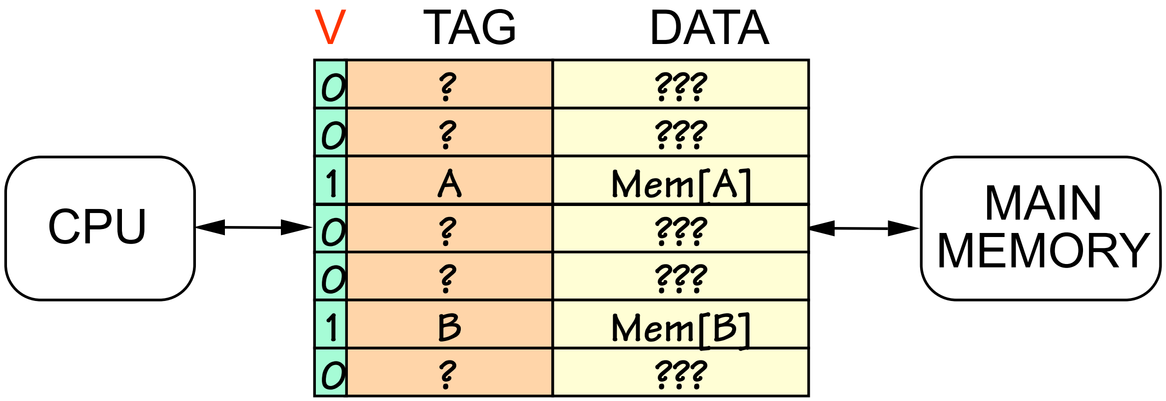
Replacement Strategy vs. Miss Rate

H&P: Figure 5.4

	Associativity					
	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

- FIFO was reported to be worse than random or LRU
- Little difference between random and LRU for larger-size caches

Valid Bits



V	TAG	DATA
0	?	???
0	?	???
1	A	Mem[A]
0	?	???
0	?	???
1	B	Mem[B]
0	?	???

Problem:

Ignoring cache lines that don't contain REAL or CORRECT values…

- on start-up
- "Back door" changes to memory (eg: loading program from disk)

Solution:

Extend each TAG with VALID bit.

- Valid bit must be set for cache line to HIT.
- On power-up / reset : clear all valid bits
- Set valid bit when cache line is FIRST *replaced.*
- Cache Control Feature: *Flush cache by clearing all valid bits*, Under program/external control.

Handling WRITES

Observation: Most (80+%) of memory accesses are READs, but writes are essential. How should we handle writes?

Policies:

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

Additional Enhancements:

- WRITE-BUFFERS: For either write-through or write-back, writes to main memory are *buffered*. CPU keeps executing while writes are completed (in order) in the background.

What combination has the highest performance?

Write-Through

```
ON REFERENCE TO Mem[X]: Look for X among tags...

HIT:            X == TAG(i) , for some cache line i

    READ:    return DATA[I]
    WRITE:   change DATA[I]; Start Write to Mem[X]

MISS: X not found in TAG of any cache line

    REPLACEMENT SELECTION:
        Select some line k to hold Mem[X]

    READ:    Read Mem[X]
             Set TAG[k] = X, DATA[k] = Mem[X]

    WRITE:   Start Write to Mem[X]
             Set TAG[k] = X, DATA[k] = new Mem[X]
```

Write-Back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
Write Back: Write Data(k) to Mem[Tag[k]]

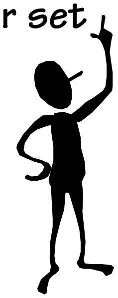
READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

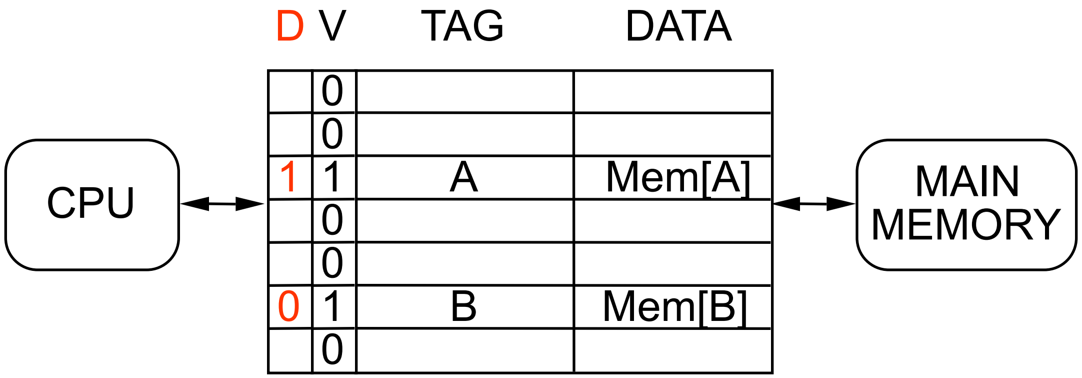
WRITE: ~~Start Write to Mem[X]~~
Set TAG[k] = X, DATA[k] = new Mem[X]

Costly if contents of cache are not modified

Write-Back w/ "Dirty" bits

Dirty and
Valid bits
are per line
not per set





D	V	TAG	DATA
	0		
	0		
1	1	A	Mem[A]
	0		
	0		
0	1	B	Mem[B]
	0		

What if the
cache has a
block-size
larger than
one word?

A) If only one
word in the line
is modified, we
end up writing
back ALL words

ON REFERENCE TO Mem[X]: Look for X among tags...

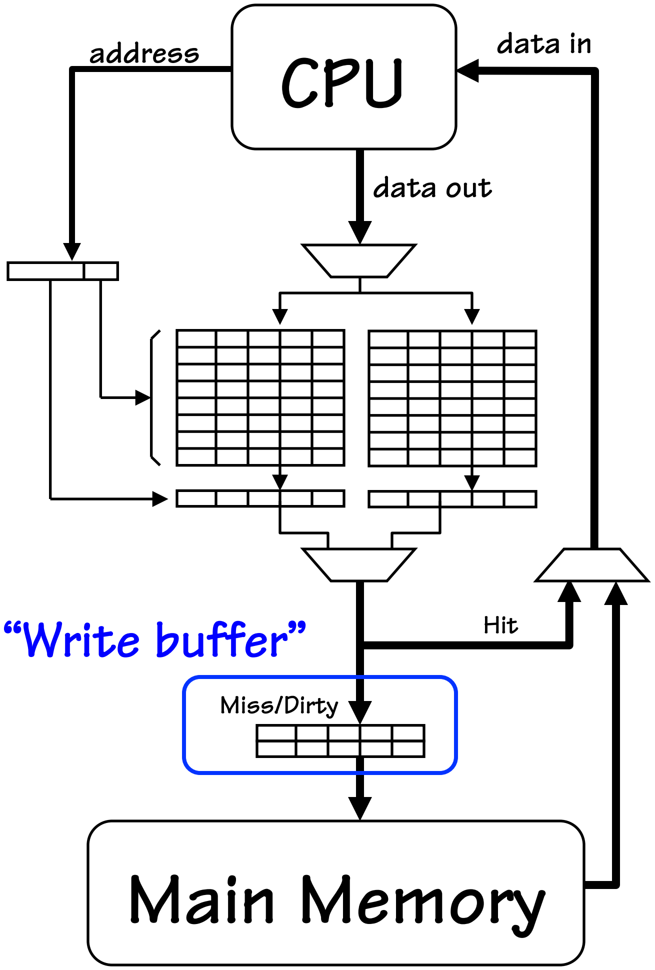
HIT: *X = TAG(i) , for some cache line I*
READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~ D[i]=1

MISS: *X not found in TAG of any cache line*
REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]
READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
WRITE: ~~Start Write to Mem[X]~~ D[k]=1 , Read Mem[X]
Set TAG[k] = X, DATA[k] = new Mem[X]

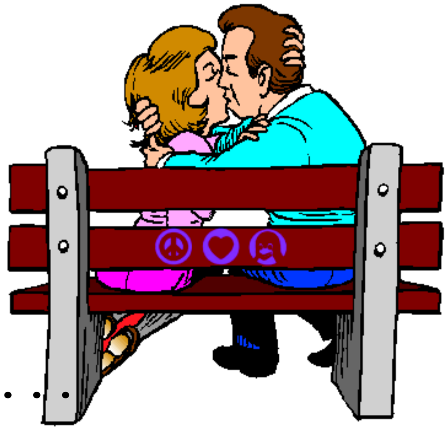
B) On a MISS, we
need to READ the
line BEFORE we
WRITE it.

Write Buffers

- Avoids the overhead of waiting for writes to complete
- Write data is stored in a special H/W queue called a "Write Buffer" where it is POSTED until the write completes
- Usually at least the size of a cache block.
- On a subsequent cache MISSes
  - you may still need to stall any subsequent reads until outstanding (POSTED) writes are completed
  - the you can check to see if the missed address matches one in the write buffer.
- Takes advantage of "sequential writes"
- Prevailing wisdom:
  - Write-Back is better than Write-Through, less memory traffic
  - Always use Write-buffering



Cache Benchmarking



Suppose this loop is entered with $t3 = 4000:

ADR:	Instruction	I	D .
`400:`	`lw   $t0,0($t3)`	`400`	`4000+...`
`404:`	`addi $t3,$t3,4`	`404`	
`408:`	`bne  $t0,$0,400`	`408`	

GOAL: Given some cache design, simulate (by hand or machine) execution well enough to estimate hit ratio.

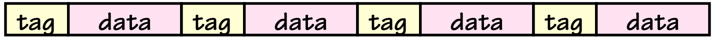
1. Observe that the sequence of memory locations referenced is
   400, 4000, 404, 408, 400, 4004, ...

We can use this simpler reference string/memory trace, rather than the program, to simulate cache behavior.

2. We can make our life even easier by converting to word addresses:
   100, 1000, 101, 102, 100, 1001, ...

(Word Addr = (Byte Addr)/4)

Simple Cache Simulation



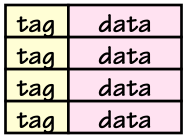
4-line Fully-associative/LRU

Addr	Line#	Miss?
100	0	M
1000	1	M
101	2	M
102	3	M
100	0	
1001	1	M
101	2	
102	3	
100	0	
1002	1	M
101	2	
102	3	
100	0	
1003	1	M
101	2	
102	3	

Compulsory Misses

Capacity Miss →

1/4 miss



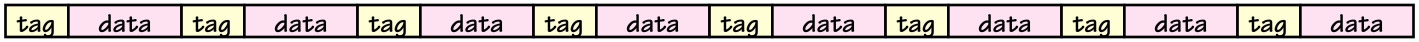
4-line Direct-mapped

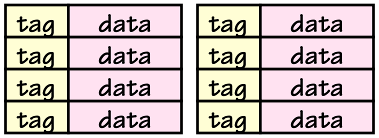
Addr	Line#	Miss?
100	0	M
1000	0	M
101	1	M
102	2	M
100	0	M
1001	1	M
101	1	M
102	2	
100	0	
1002	2	M
101	1	
102	2	M
100	0	
1003	3	M
101	1	
102	2	

← Collision Miss

7/16 miss

Cache Simulation: Bout 2





8-line Fully-associative, LRU

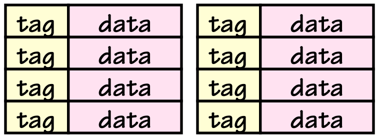
Addr	Line#	Miss?
100	0	M
1000	1	M
101	2	M
102	3	M
100	0	
1001	4	M
101	2	
102	3	
100	0	
1002	5	M
101	2	
102	3	
100	0	
1003	6	M
101	2	
102	3	

1/4 miss

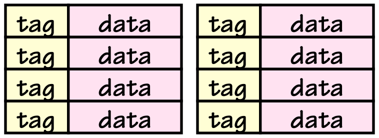
2-way, 8-line total, LRU

Addr	Line/N	Miss?
100	0,0	M
1000	0,1	M
101	1,0	M
102	2,0	M
100	0,0	
1001	1,1	M
101	1,0	
102	2,0	
100	0,0	
1002	2,1	M
101	1,0	
102	2,0	
100	0,0	
1003	3,0	M
101	1,0	
102	2,0	

1/4 miss



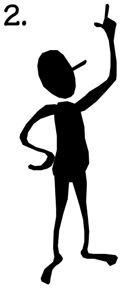
Cache Simulation: Bout 3



2-way, 8-line total, LRU

Addr	Line/N	Miss?
100	0,0	
1004	0,1	M
101	1,0	
102	2,0	
100	0,0	
1005	1,1	M
101	1,0	
102	2,0	
100	0,0	
1006	2,1	M
101	1,0	
102	2,0	
100	0,0	
1007	3,1	M
101	1,0	
102	2,0	

The first 16 cycles of both caches are identical (Same as 2-way on previous slide). So we jump to round 2.



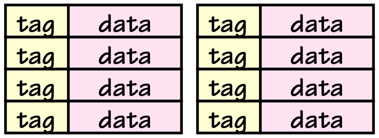
1/4 miss

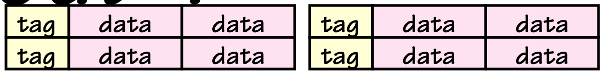
2-way, 8-line total, FIFO

Addr	Line/N	Miss?
100	0,0	
1004	0,0	M
101	1,0	
102	2,0	
100	0,1	M
1005	1,0	M
101	1,1	M
102	2,0	
100	0,0	
1006	2,0	M
101	1,0	
102	2,1	M
100	0,0	
1007	3,1	M
101	1,0	
102	2,0	

7/16 miss

Cache Simulation: Bout 4





2-way, 8-line total, LRU

Addr	Line/N	Miss?
100	0,0	M
1000	0,1	M
101	1,0	M
102	2,0	M
100	0,0	
1001	1,1	M
101	1,0	
102	2,0	
100	0,0	
1002	2,1	M
101	1,0	
102	2,0	
100	0,0	
1003	3,0	M
101	1,0	
102	2,0	

1/4 miss

2-way, 4-line, 2 word blk, LRU

Addr	Line/N	Miss?
100/1	0,0	M
1000/1	0,1	M
101	0,0	
102/3	1,0	M
100	0,0	
1001	0,1	
101	0,0	
102	1,0	
100	0,0	
1002/3	1,1	M
101	0,0	
102	1,0	
100	0,0	
1003	1,1	
101	0,0	
102	1,0	

1/8 miss

Cache Design Summary

- Various design decisions the affect cache performance
  - Block size, exploits spatial locality, saves tag H/W, but, if blocks are too large you can load unneeded items at the expense of needed ones
  - Replacement strategy, attempts to exploit temporal locality to keep frequently referenced items in cache
    - LRU – Best performance/Highest cost
    - FIFO – Low performance/Economical
    - RANDOM – Medium performance/Lowest cost, avoids pathological sequences, but performance can vary
  - Write policies
    - Write-through – Keeps memory and cache consistent, but high memory traffic
    - Write-back – allows memory to become STALE, but reduces memory traffic
    - Write-buffer – queue that allows processor to continue while waiting for writes to finish, reduces stalls
- No simple answers, in the real-world cache designs are based on simulations using memory traces.