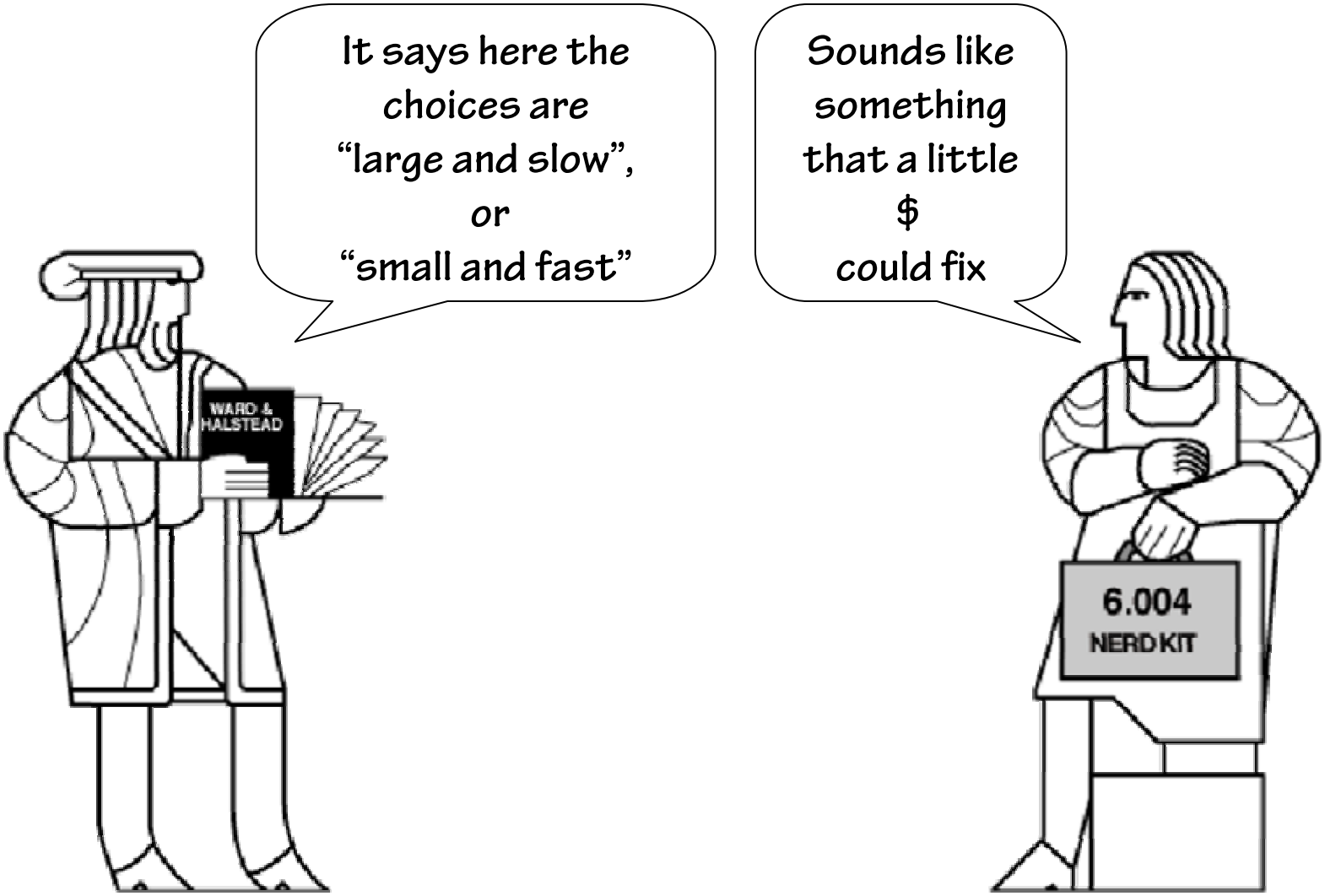
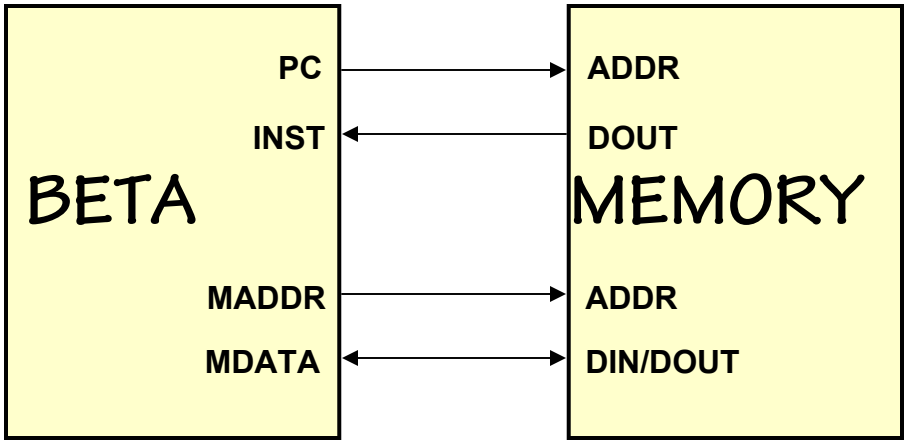
The Memory Hierarchy



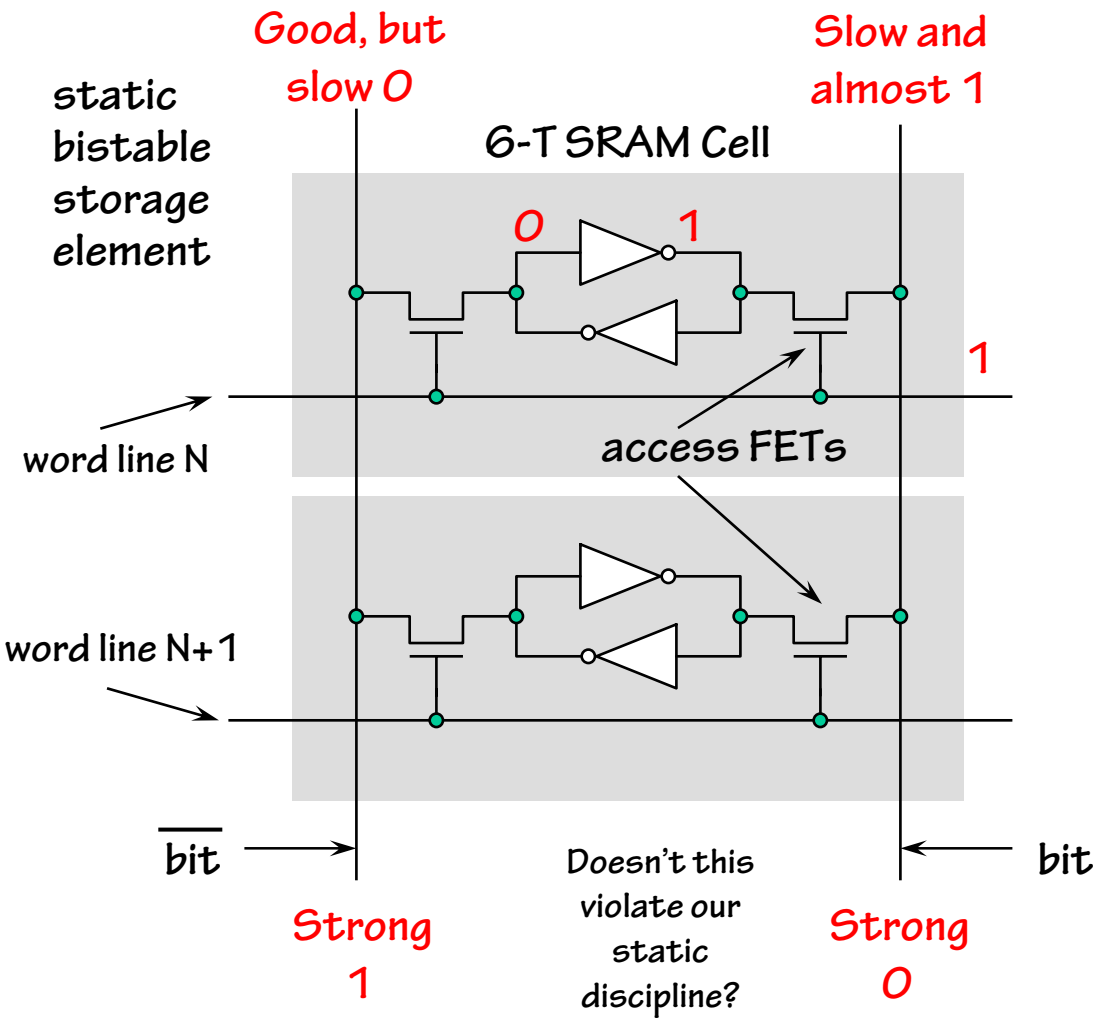
What we want in a memory



	Capacity	*Latency*	*Cost*
Register	100's of bits	20 ps	$$$$
SRAM	100' Kbytes	1 ns	$$$
DRAM	100' Mbytes	40 ns	$
Hard disk*	10's Gbytes	10 ms	¢
Want	100 Mbytes	1 ns	cheap

* non-volatile

SRAM Memory Cell



There are two bit-lines per column, one supplies the bit the other it's complement.

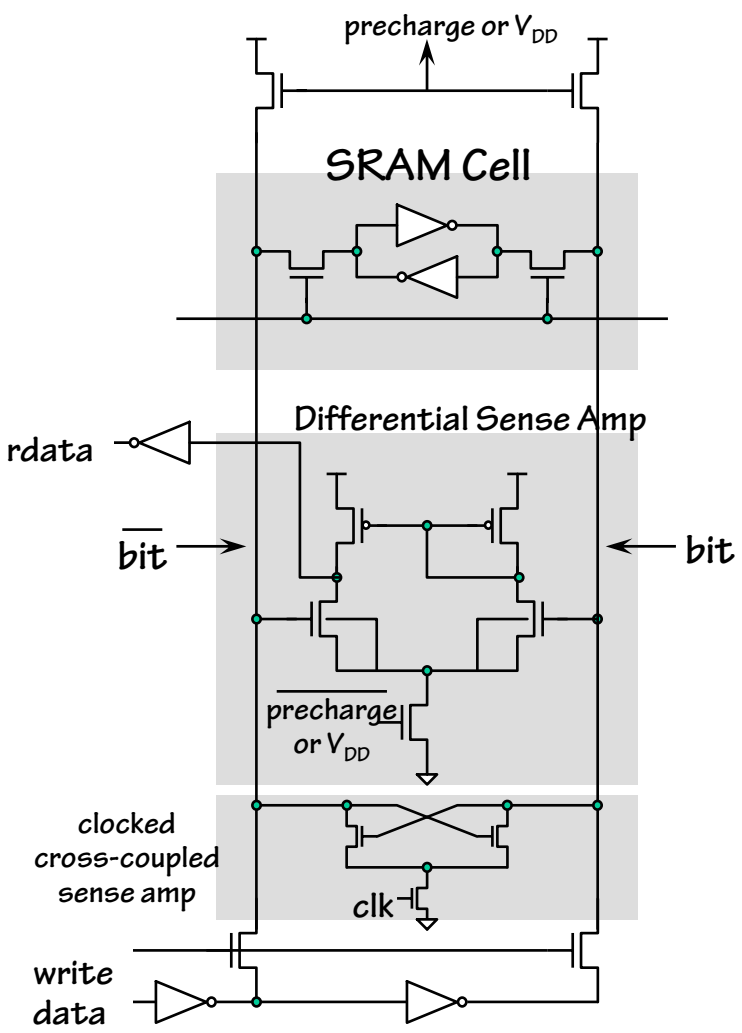
On a Read Cycle -

A single word line is activated (driven to "1"), and the access transistors enable the selected cells, and their complements, onto the bit lines.

Writes are similar to reads, except the bit-lines are driven with the desired value of the cell.

The writing has to "overpower" the original contents of the memory cell.

Tricks to make SRAMs fast

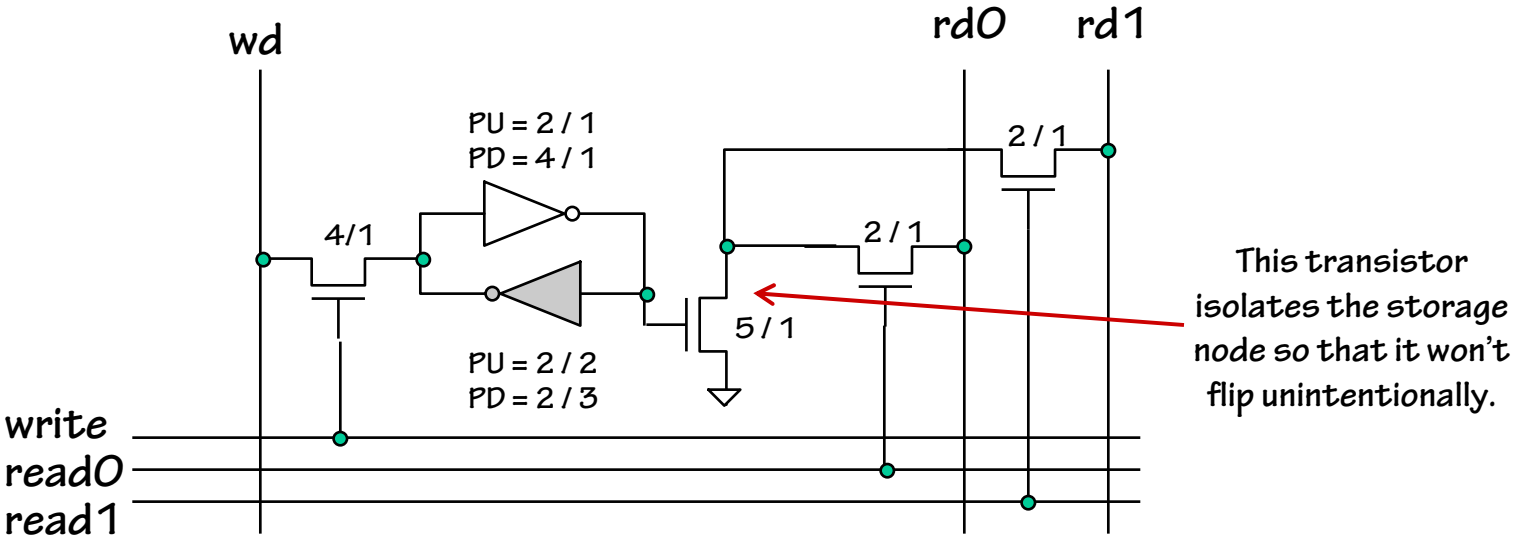


Forget that it is a digital circuit

1) Precharge the bit lines prior to the read (for instance- while the address is being decoded) because the access FETs are good pull-downs and poor pull-ups

2) Use a differential amplifier to "sense" the difference in the two bit-lines long before they reach a valid logic levels.

Multiport SRAMs (a.k.a. Register Files)

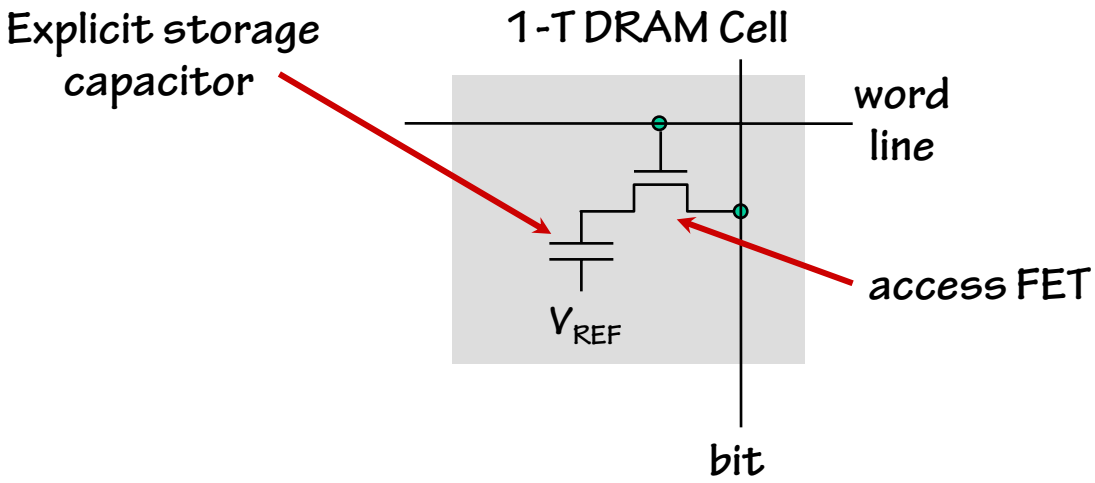


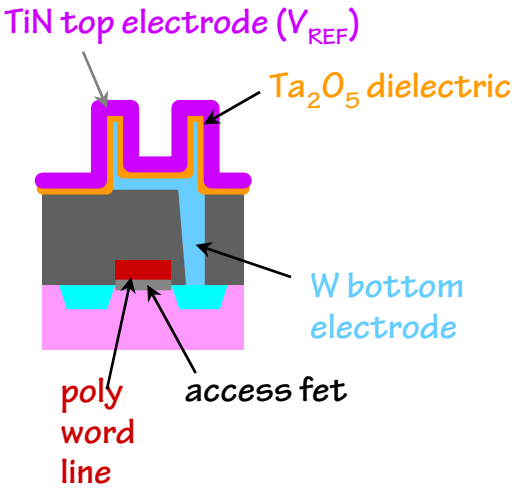
One can increase the number of SRAM ports by adding access transistors. By carefully sizing the inverter pair, so that one is strong and the other weak, we can assure that our WRITE bus will only fight with the weaker one, and the READs are driven by the stronger one. Thus minimizing both access and write times.

What is the cost per cell of adding a new read or write port?

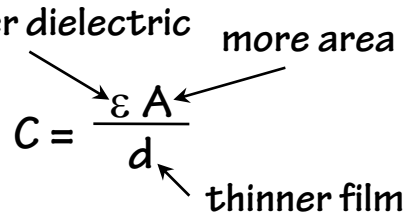
1-T Dynamic Ram

Six transistors/cell may not sound like much, but they can add up quickly. What is the fewest number of transistors that can be used to store a bit?



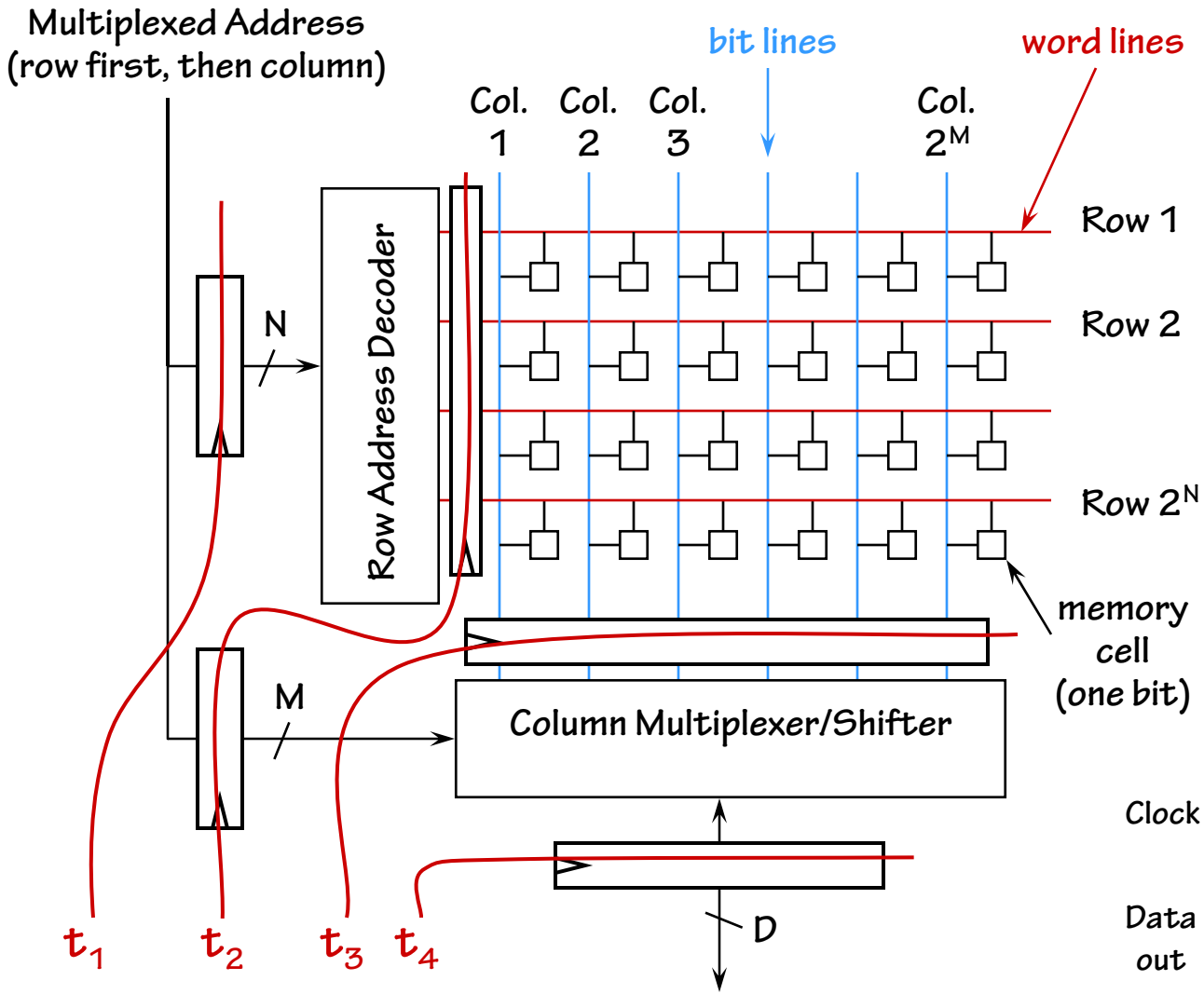


C in storage capacitor determined by:



Tricks for increasing throughput

but, alas, not latency

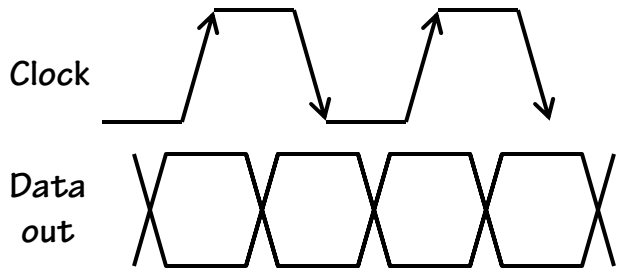


The first thing that should pop into you mind when asked to speed up a digital design…

PIPELINING

Synchronous DRAM (SDRAM)

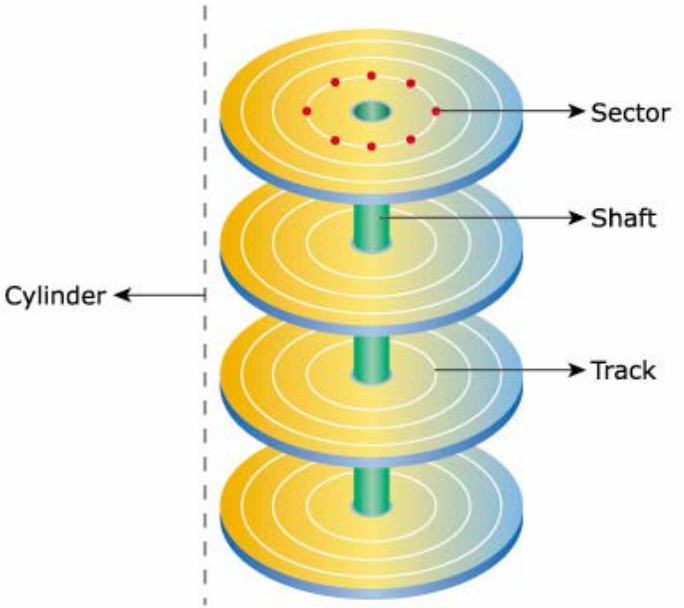
Double-clocked Synchronous DRAM (DDRAM)

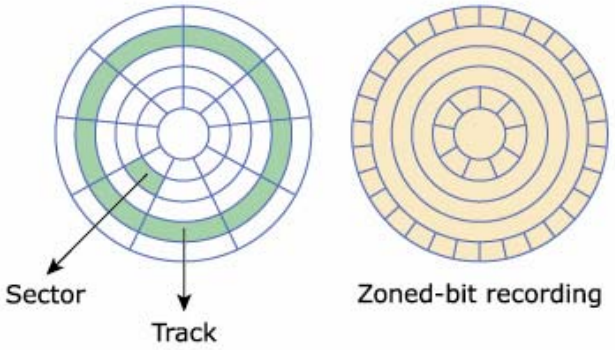


Hard Disk Drives

Typical high-end drive:

- Average latency = 4 ms
- Average seek time = 9 ms
- Transfer rate = 20M bytes/sec
- Capacity = 60G byte
- Cost = ~~$180~~ $99



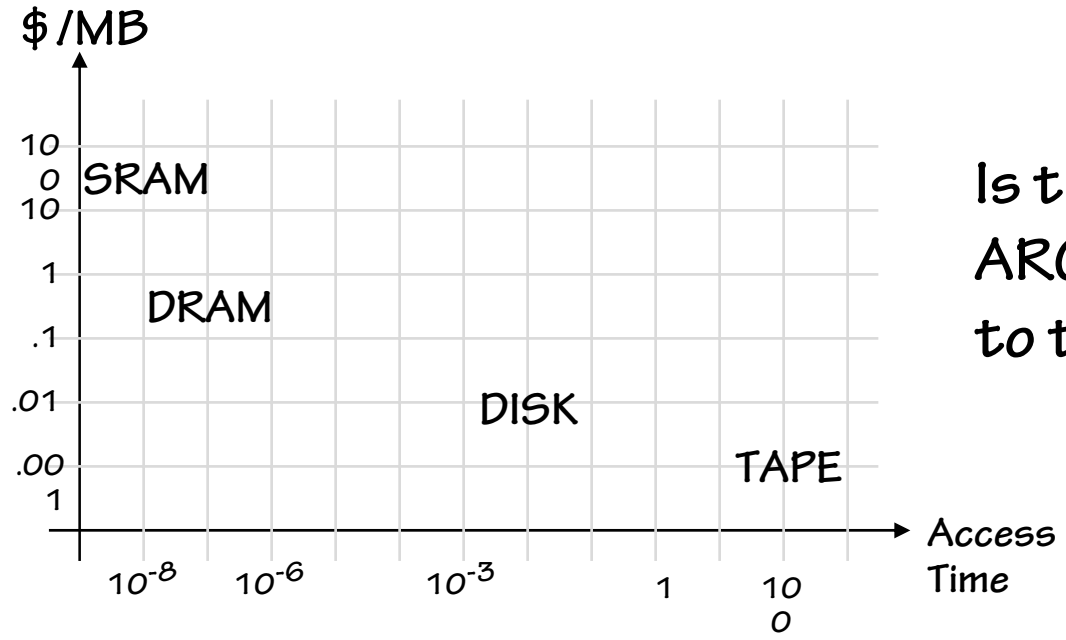


figures from www.pctechguide.com

Quantity vs Quality…

Your memory system can be
- BIG and SLOW… or
- SMALL and FAST.

We've explored a range of circuit-design trade-offs.



Is there an ARCHITECTURAL solution to this DILEMMA?

Best of Both Worlds

What we WANT: A BIG, FAST memory!

We'd like to have a memory system that

- PERFORMS like 32 MBytes of SRAM; but
- COSTS like 32 MBytes of slow memory.

SURPRISE: We can (nearly) get our wish!

KEY: Use a hierarchy of memory technologies:

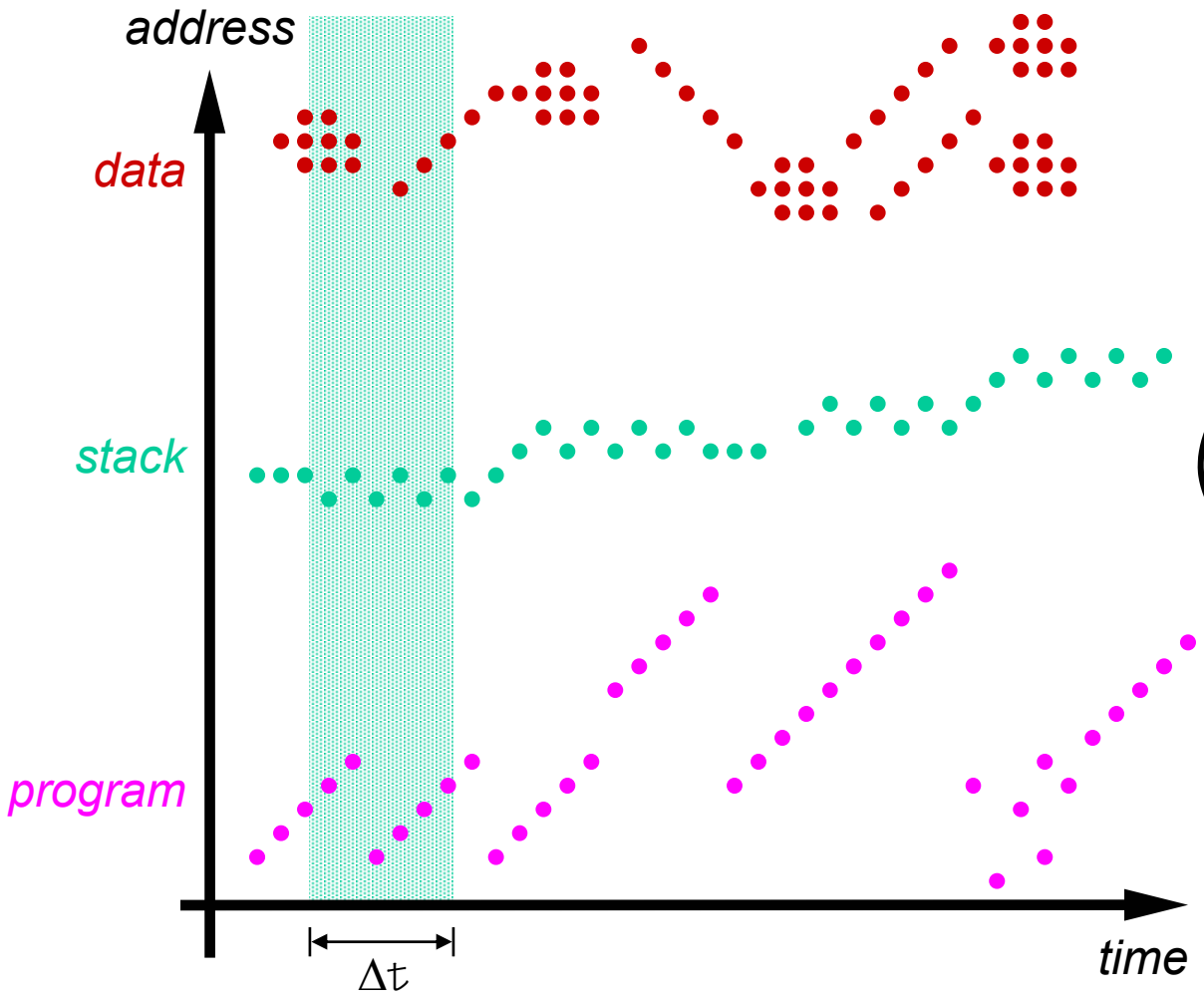
Key IDEA

- Keep the most often-used data in a small, fast SRAM (often local to CPU chip)
- Refer to Main Memory only rarely, for remaining data.
- The reason this strategy works: LOCALITY

Locality of Reference:

Reference to location X at time t implies that reference to location $X+\Delta X$ at time $t+\Delta t$ becomes more probable as ΔX and Δt approach zero.

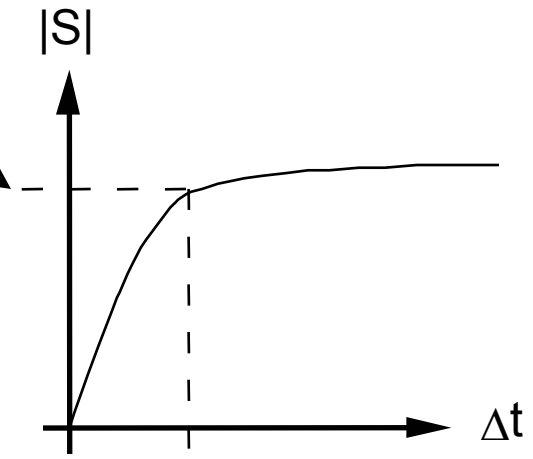
Memory Reference Patterns



S is the set of locations accessed during Δt.

Working set: a set S which changes slowly wrt access time.

Working set size, |S|



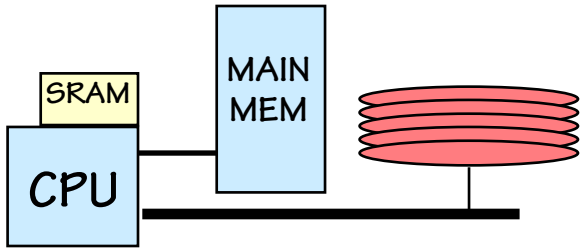
Exploiting the Memory Hierarchy

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

- Registers, Main Memory,

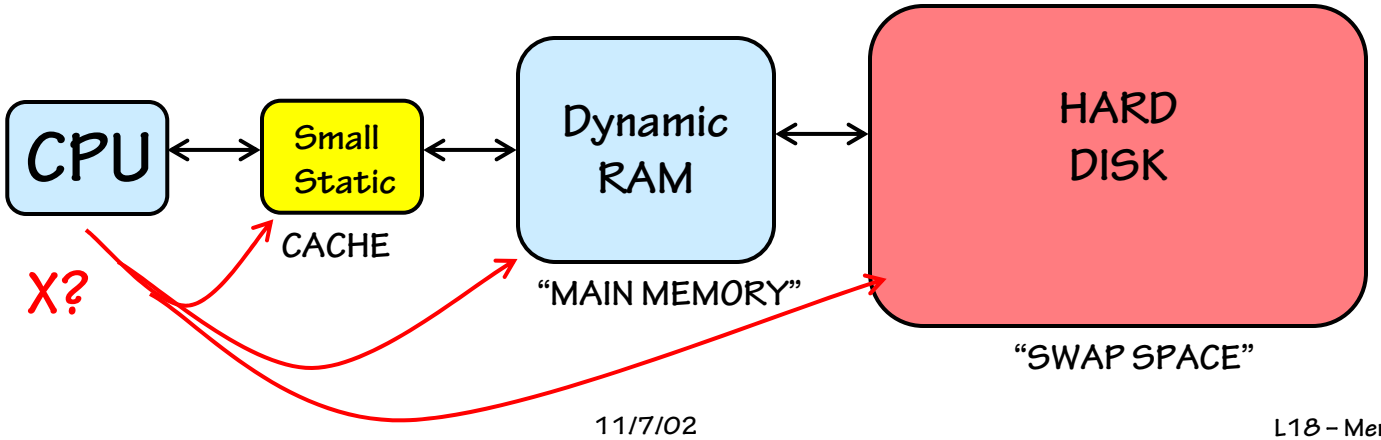
  Disk each available as
  storage alternatives;

- Tell programmers: "Use them cleverly"



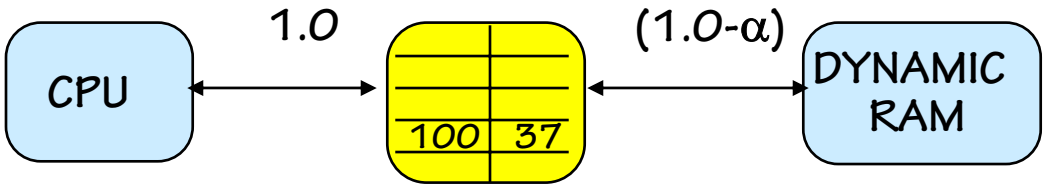
Approach 2: *Hide* Hierarchy

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



The Cache Idea:

Program-Transparent Memory Hierarchy



Cache contains TEMPORARY COPIES of selected main memory locations... eg. Mem[100] = 37

GOALS:

1) Improve the *average access* time

α HIT RATIO: Fraction of refs found in CACHE.

$(1-\alpha)$ MISS RATIO: Remaining references.

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

2) Transparency (compatibility, programming ease)

Challenge: make the hit ratio as high as possible.

How High of a Hit Ratio?

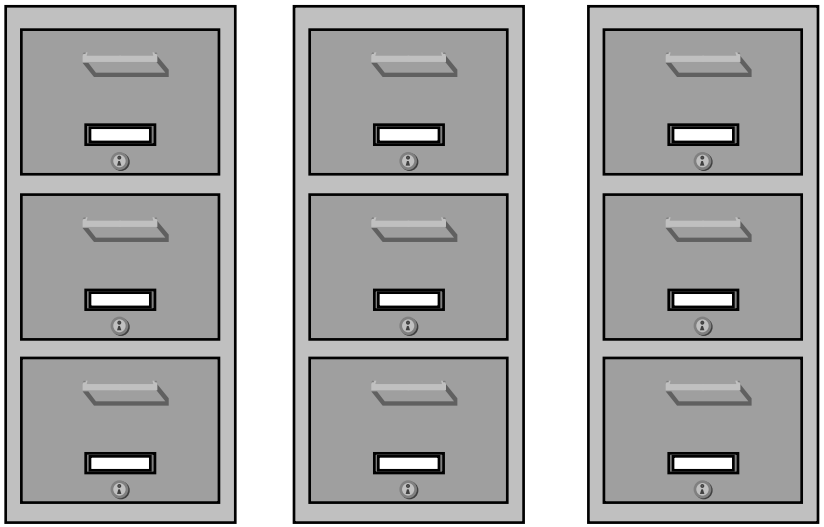
Suppose we can easily build an on-chip static memory with a 4 nS access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 40 nS. How high of a hit rate do we need to sustain an average access time of 5 nS?

$$\alpha = 1 - \frac{t_{ave} - t_c}{t_m} = 1 - \frac{5 - 4}{40} = 97.5\%$$

The Cache Principle

Find "Bitdiddle, Ben"

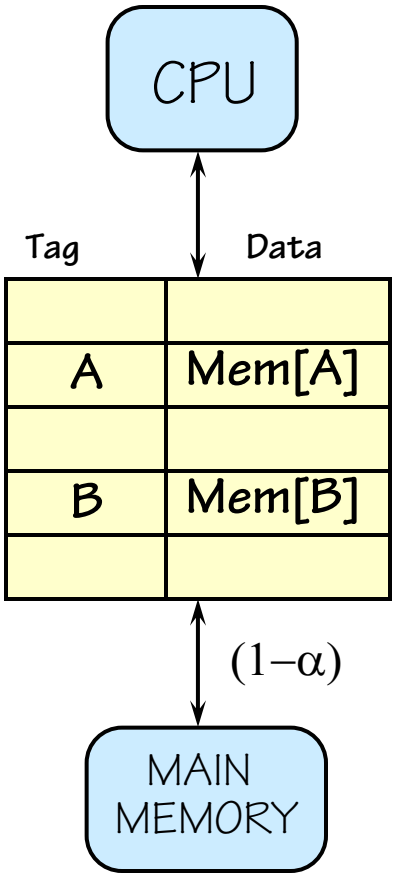
5-Minute Access Time:



5-Second Access Time:

ALGORITHM: Look nearby for the requested information first, if its not there check secondary storage

Basic Cache Algorithm



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

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

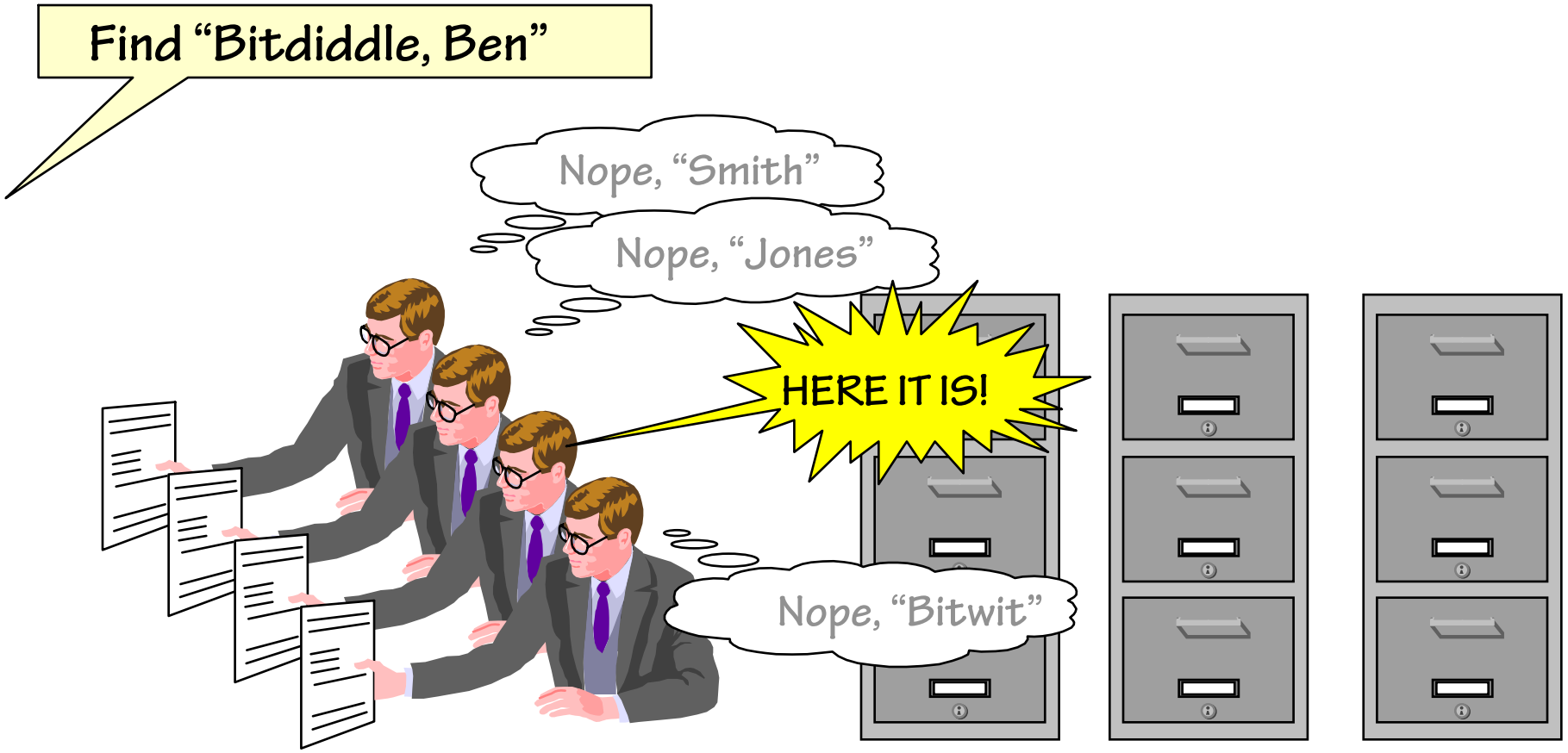
- READ: return DATA(i)
- WRITE: change DATA(i); Start Write to Mem(X)

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

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X] (Allocation)
- 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]

QUESTION: How do we "search" the cache?

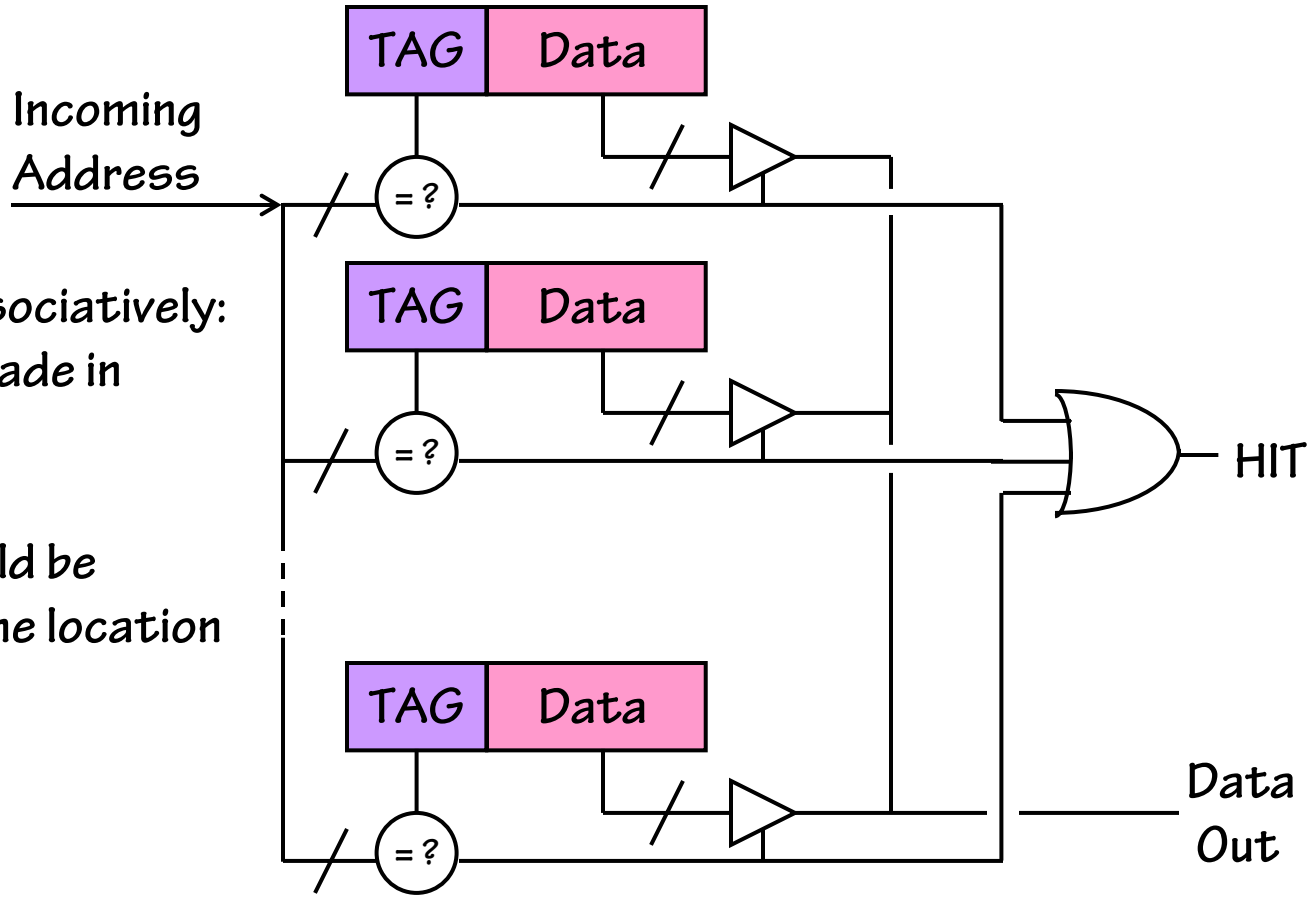
Associativity: Parallel Lookup



Fully-Associative Cache

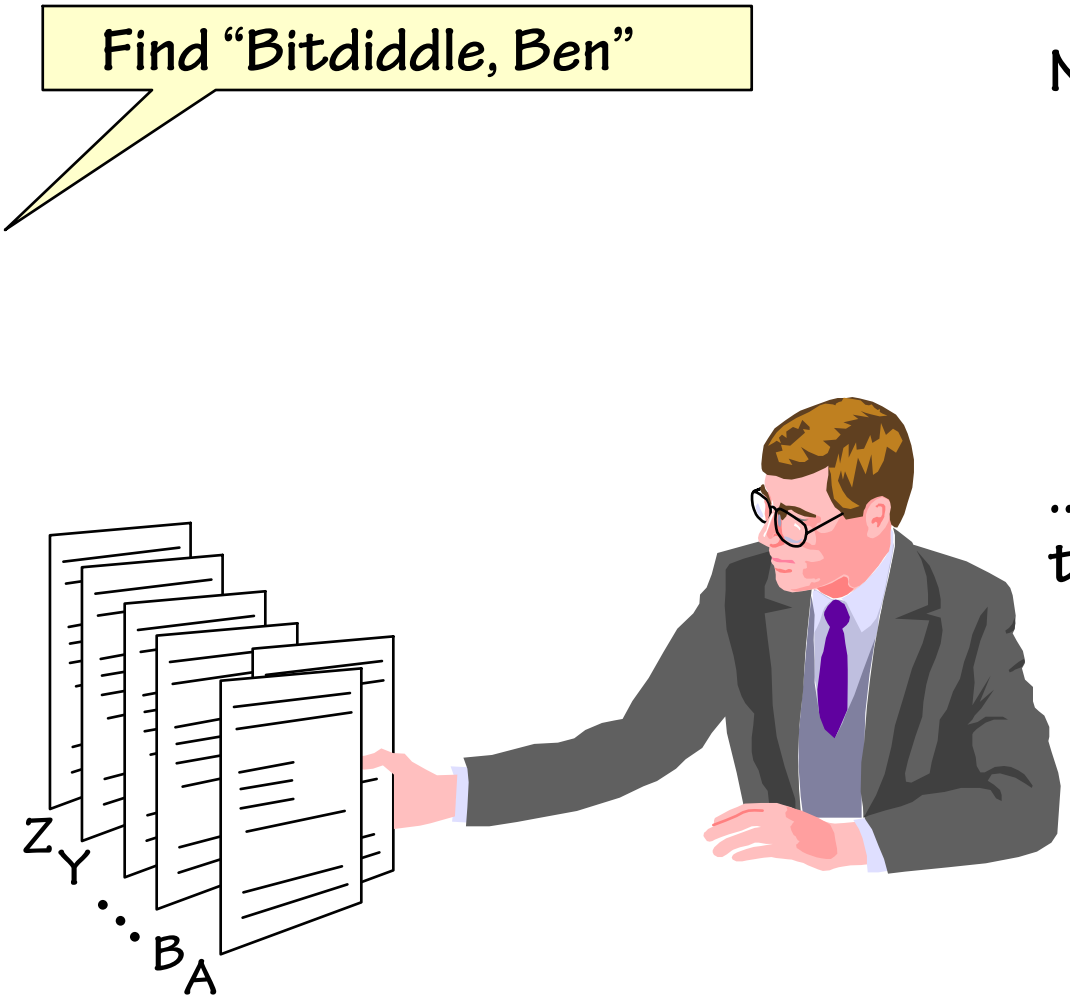
The extreme in associatively: All comparisons made in parallel

Any data item could be located in any cache location



Direct-Mapped Cache

(non-associative)

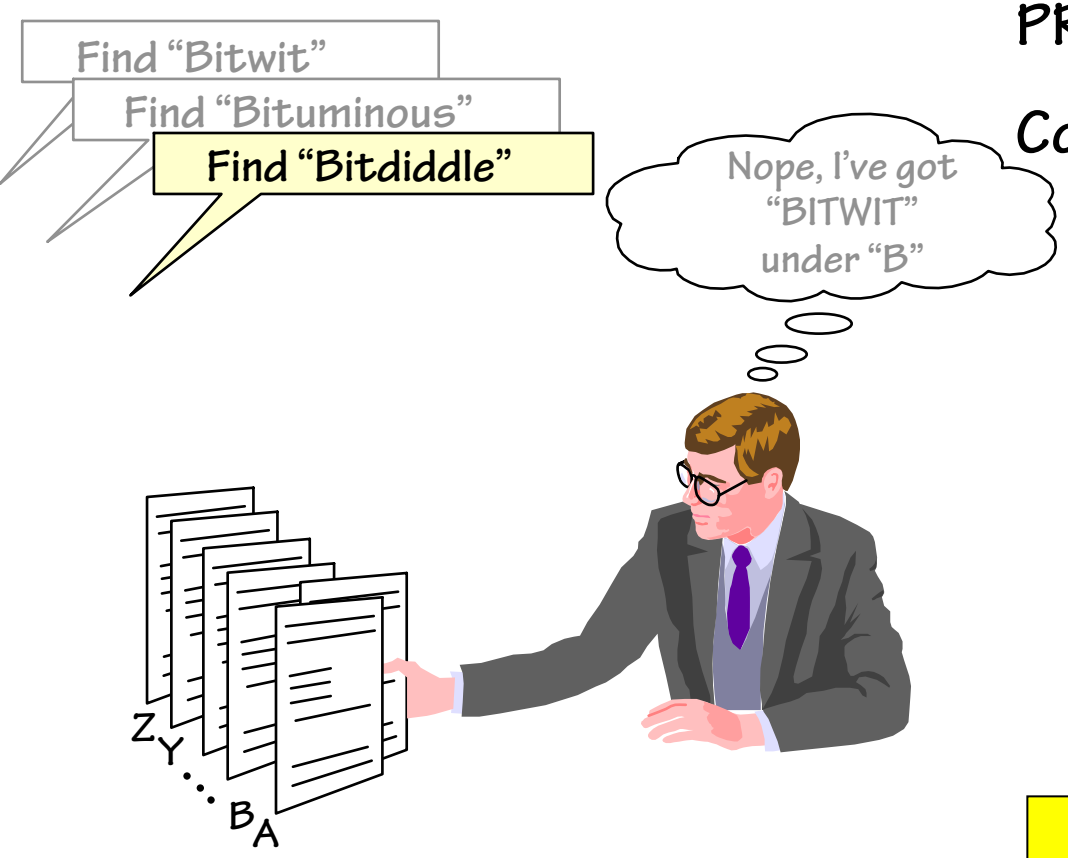


NO Parallelism:

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

... can use ordinary RAM as table

The Problem with Collisions



PROBLEM:

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

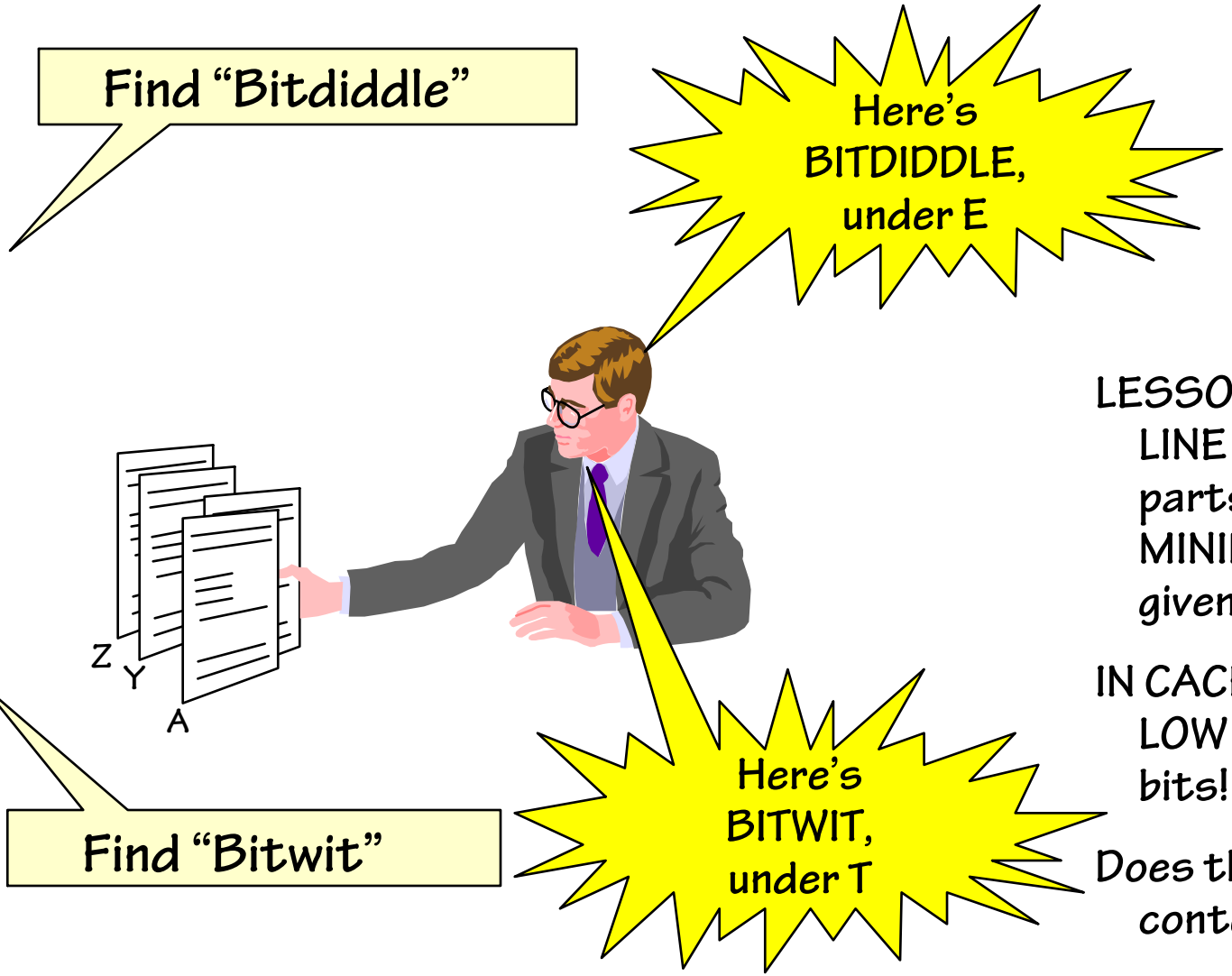
- CAN'T cache both "Bitdiddle" & "Bitwit"

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

BETTER IDEA:
File by LAST letter!

Optimizing for Locality:

selecting on statistically independent bits



LESSON: Choose CACHE LINE from *independent* parts of request to MINIMIZE CONFLICT given locality patterns...

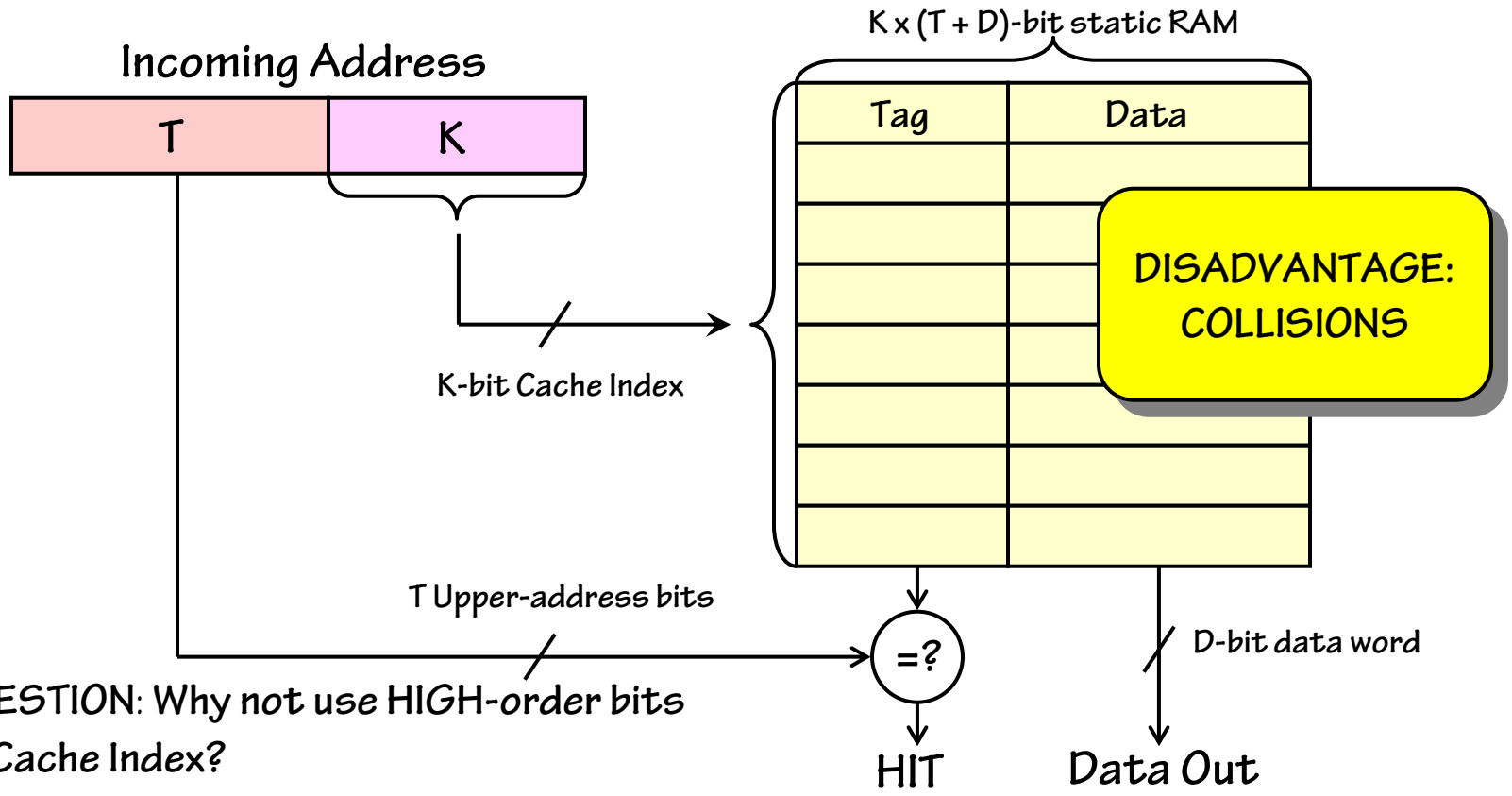
IN CACHE: Select line by LOW ORDER address bits!

Does this ELIMINATE contention?

Direct Mapped Cache

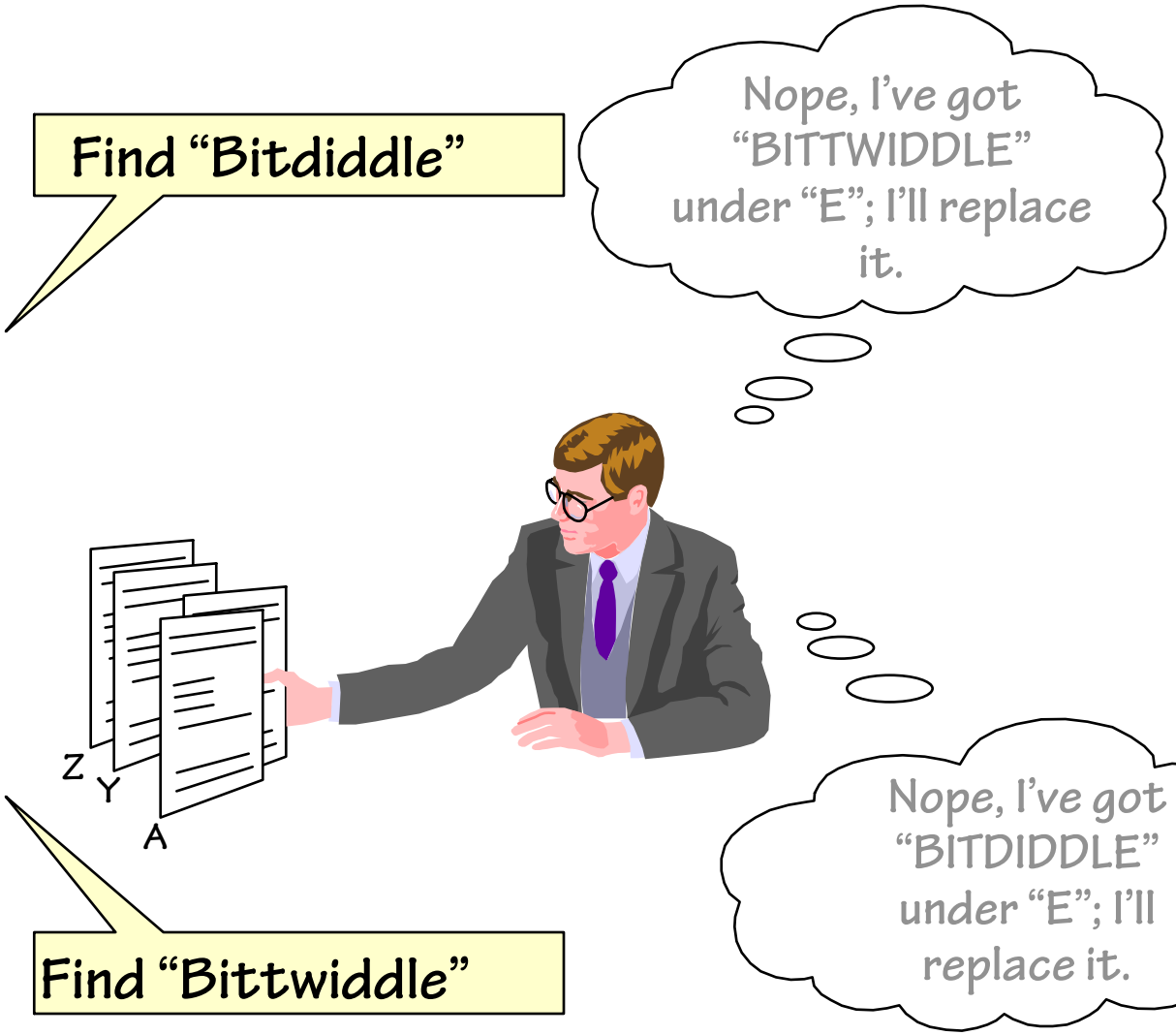
Low-cost extreme:

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



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

Contention, Death, and Taxes...



LESSON: In a non-associative cache, SOME pairs of addresses must compete for cache lines...

... if working set includes such pairs, we get THRASHING and poor performance.

Direct-Mapped Cache Contention

Loop A: Pgm at 1024, data at 37:

Memory Address	Cache Line	Hit/ Miss
1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
...		

Loop B: Pgm at 1024, data at 2048:

1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
...		

Works GREAT here…

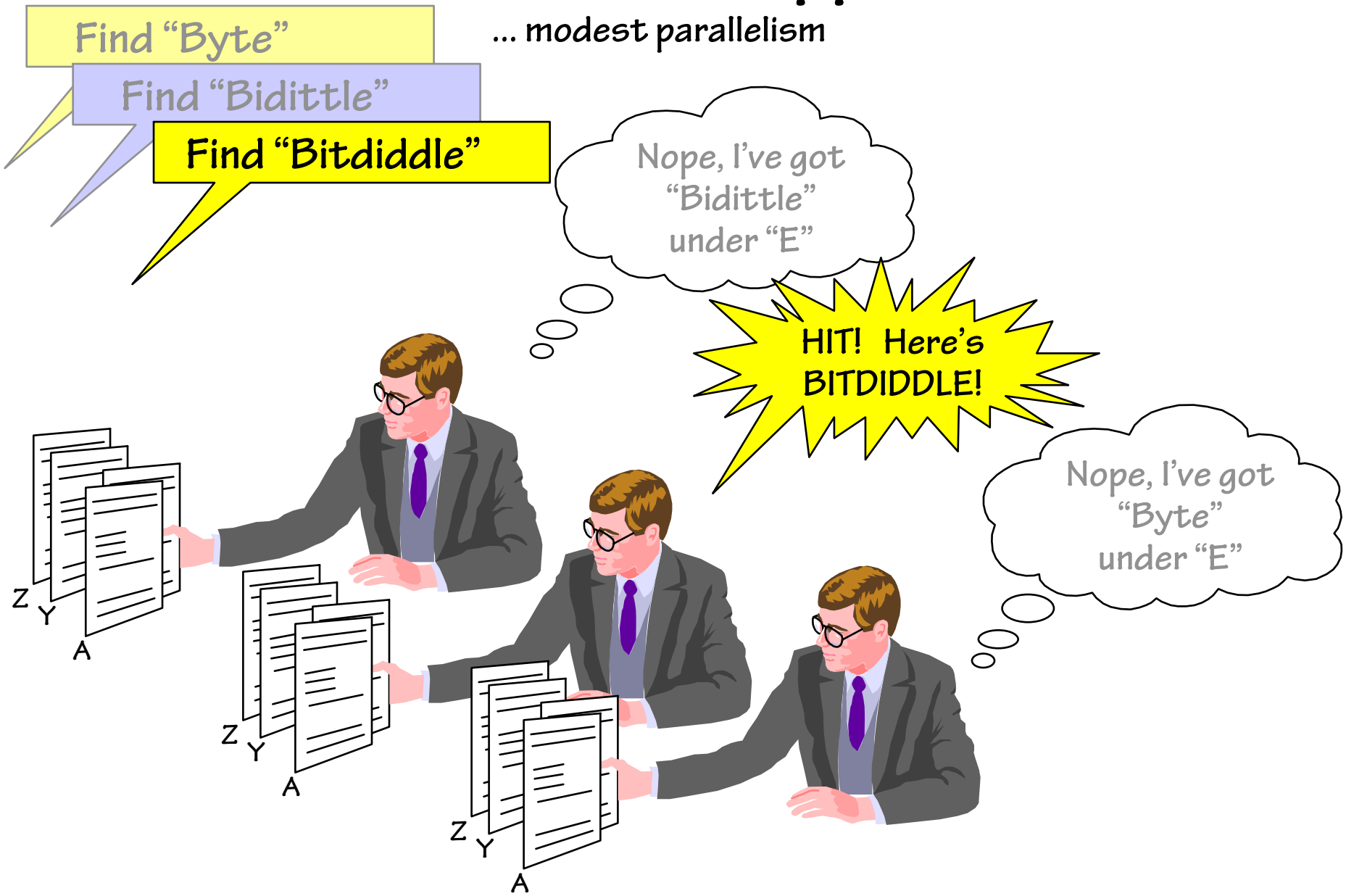
Assume 1024-line direct-mapped cache, 1 word/line. Consider tight loop, at steady state:
(assume WORD, not BYTE, addressing)

… but not here!

We need *some* associativity, But not *full* associativity…

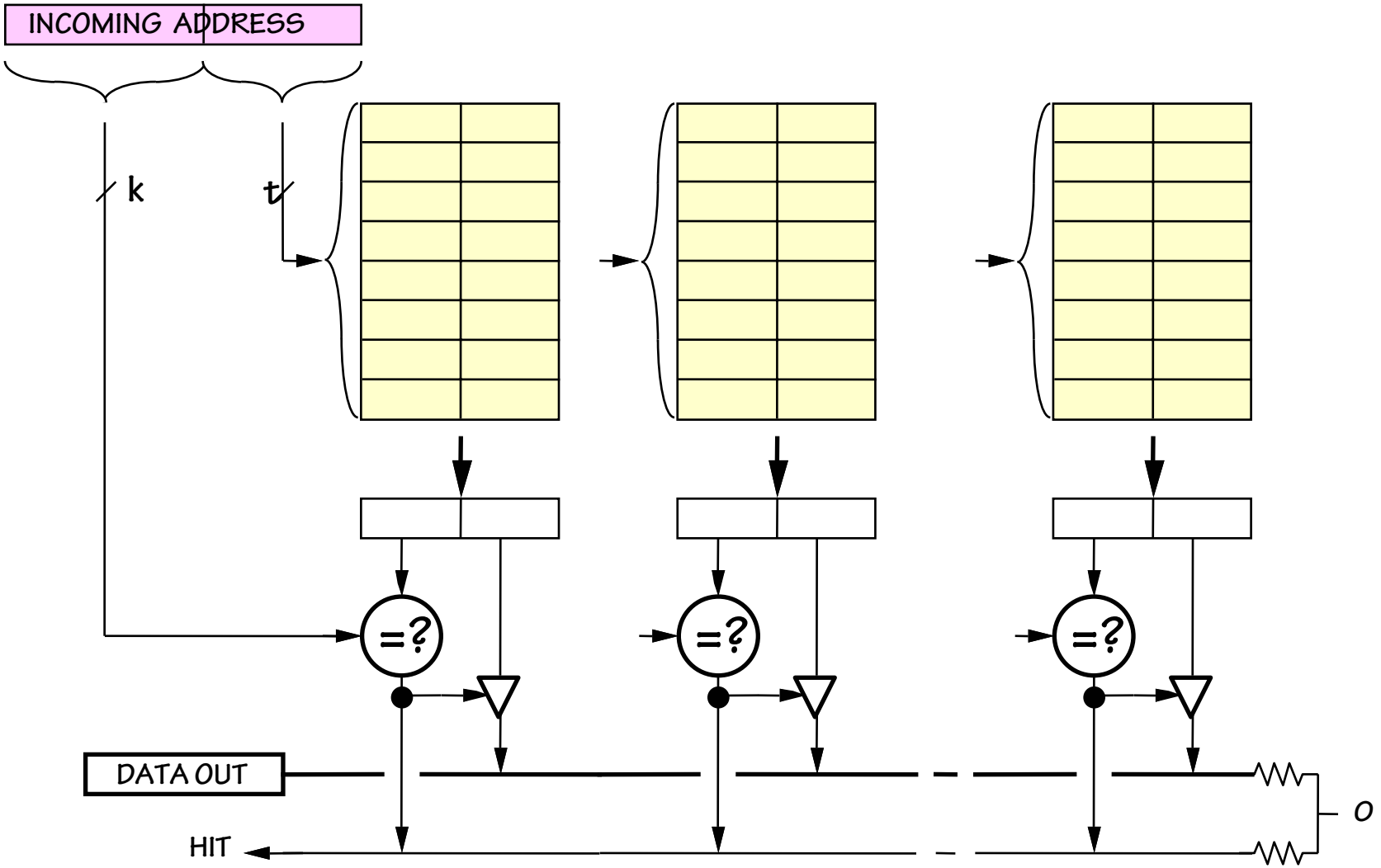
Set Associative Approach...

... modest parallelism



N-way Set-Associative Cache

Can store N colliding entries at once!



Things to Cache

- What we've got: basic speed/cost tradeoffs.
- Need to exploit a hierarchy of technologies
- Key: Locality. Look for "working set", keep in fast memory.
- Transparency as a goal
- Transparent caches: hits, misses, hit/miss ratios
- Associativity: performance at a cost. Data points:
  - Fully associative caches: no contention, prohibitive cost.
  - Direct-mapped caches: mostly just fast RAM. Cheap, but has contention problems.
  - Compromise: set-associative cache. Modest parallelism handles contention between a few overlapping "hot spots", at modest cost.