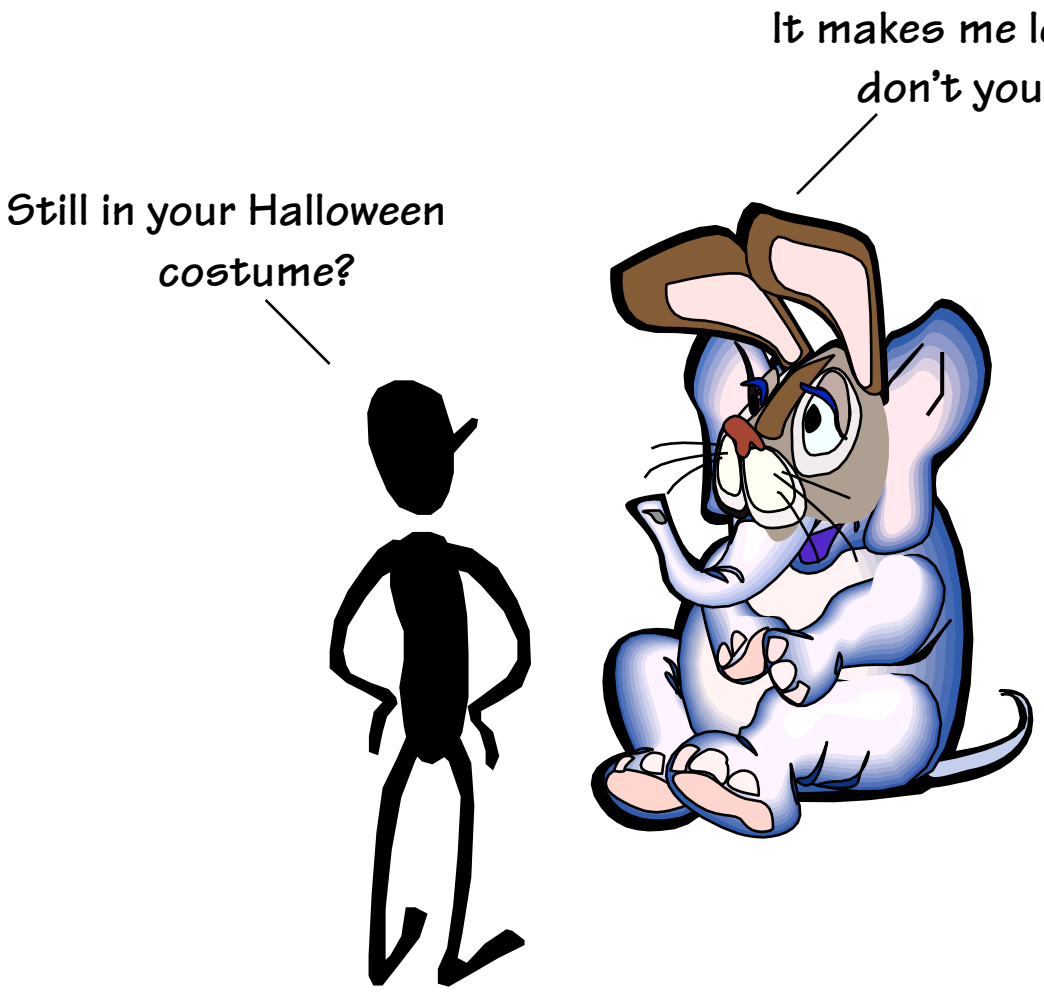
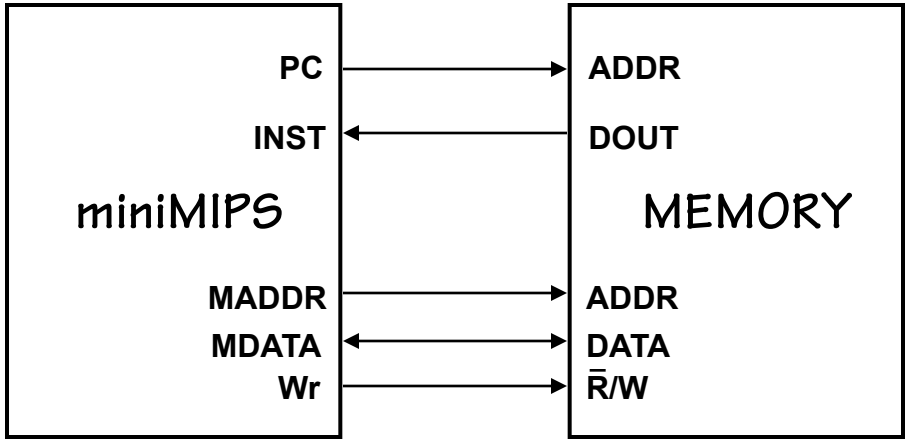
Memory Hierarchy



- Memory Flavors
- Principle of Locality
- Program Traces
- Memory Hierarchies
- Associativity

Midterm #2 Study Session Tomorrow (11/13) during lab.

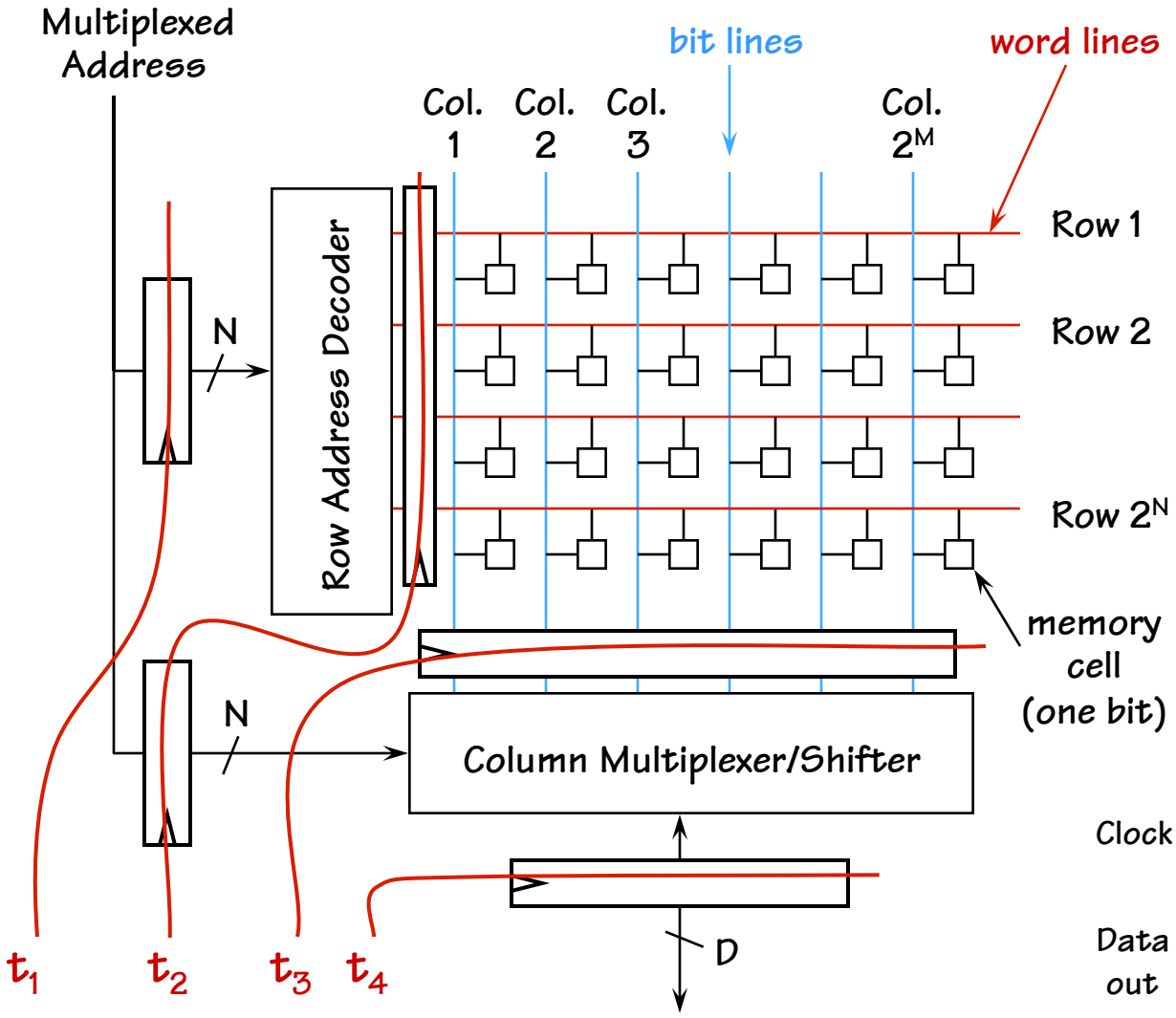
What Do We Want in a Memory?



	Capacity	*Latency*	*Cost*
Register	1000's of bits	10 ps	$$$$
SRAM	100's Kbytes	0.2 ns	$$$
DRAM	100's Mbytes	5 ns	$
Hard disk*	10's Tbytes	10 ms	¢
Want?	4 Gbyte	0.2 ns	cheap

* non-volatile

Tricks for Increasing Throughput



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

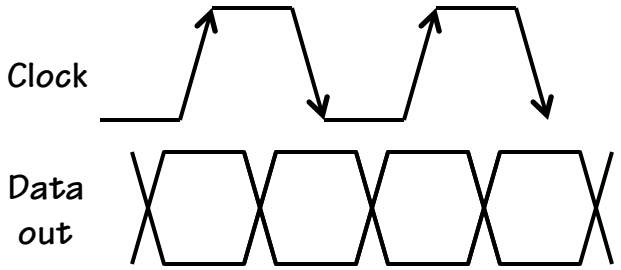
PIPELINING

Synchronous DRAM (SDRAM)

20nS reads and writes

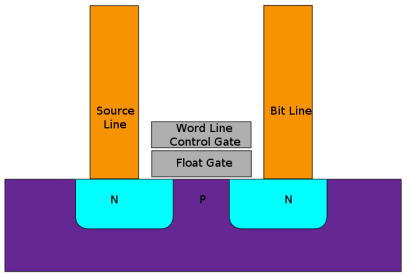
($5 per Gbyte)

Double Data Rate Synchronous DRAM (DDR)



Solid-State Disks

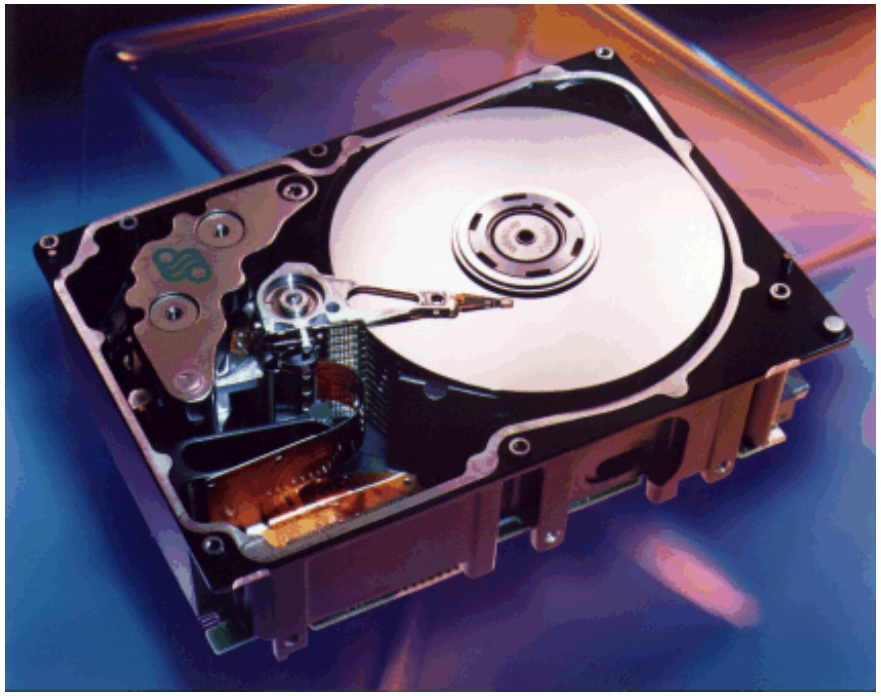
Modern solid-state disks are a non-volatile (they don't forget their contents when powered down) alternative to dynamic memory. They use a special type of "floating-gate" transistor to store data. This is done by applying a electric field large enough to actually cause carriers (ions) to permanently migrate into the gate, thus turning the switch (bit) permanently on. They are, however, not ideally suited for "main memory". Reasons:

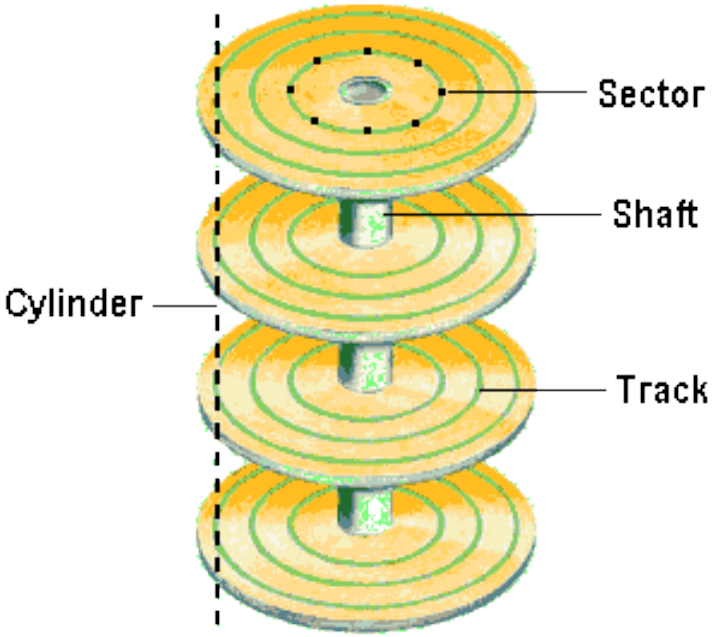


- They tend not to be randomly addressable. You can only access data in large blocks, and you need to sequentially scan through the block to get a particular value.
- Asymmetric read and write times. Writes are often 10x-20x slower than reads.
- The number of write cycles is limited (Practically 10^7-10^9, which seems like a lot for saving images, but a single variable might be written that many times in a normal program), and writes are generally an entire block at a time.

300ns read + latency
6000ns write + latency
($1 per Gbyte)

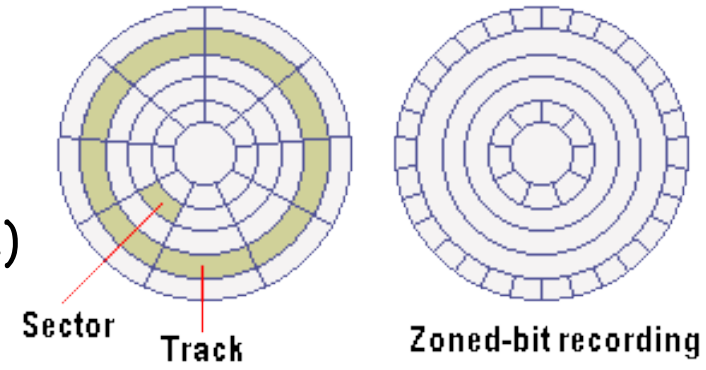
Traditional Hard Disk Drives





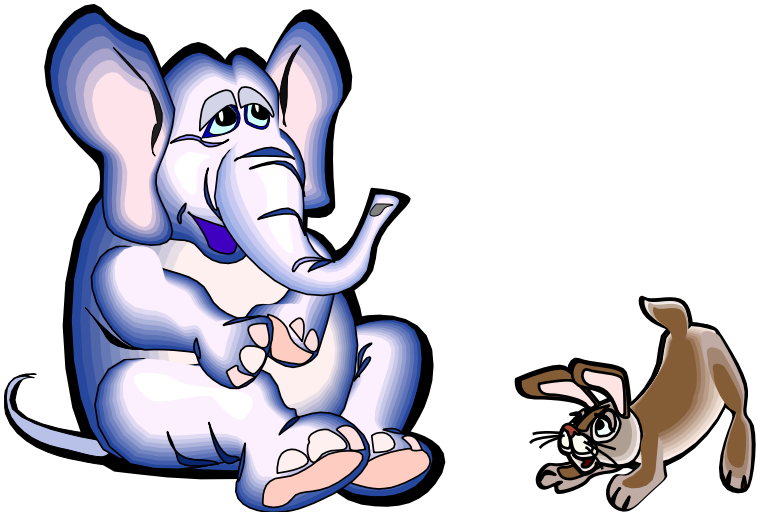
Typical high-end drive:

- Average seek time = 8.5 ms
- Average latency = 4 ms (7200 rpm)
- Transfer rate = 300 Mbytes/s (SATA)
- Capacity = 2000 G byte
- Cost = $100 (5¢ Gbyte)



figures from www.pctechguide.com

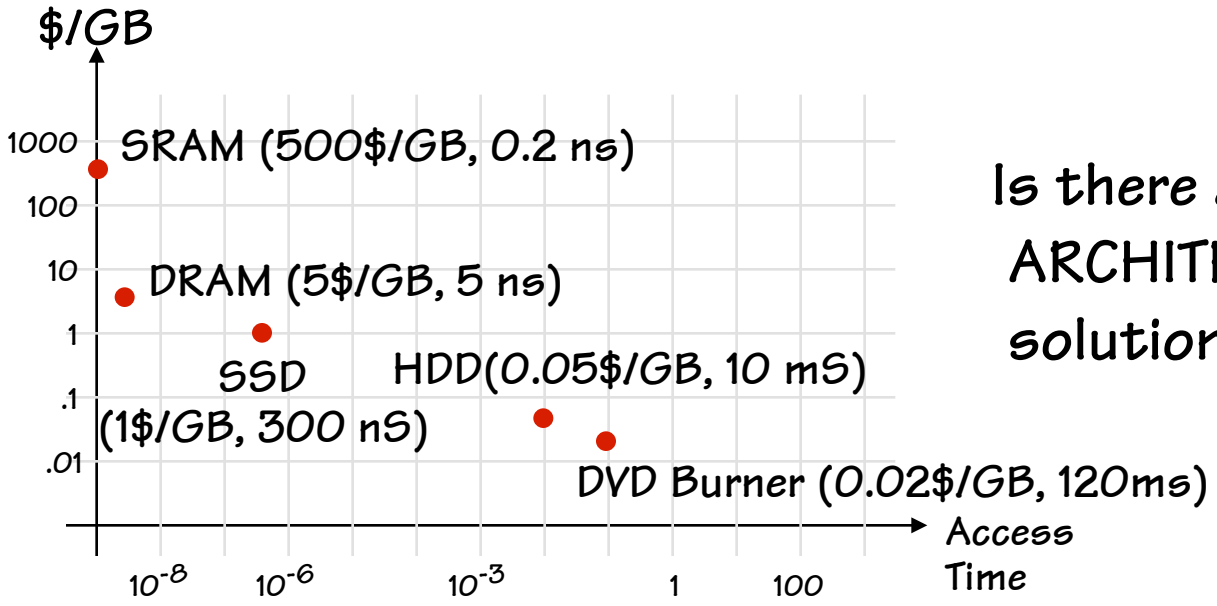
Quantity vs Quality…



Memory systems can be either:

- BIG and SLOW… or
- SMALL and FAST.

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



Is there an ARCHITECTURAL solution to this DELIMA?

Managing Memory via Programming

- In reality, systems are built with a mixture of all these various memory types

- How do we make the most effective use of each memory?
- We could push all of these issues off to programmers
  - Keep most frequently used variables and stack in SRAM
  - Keep large data structures (arrays, lists, etc) in DRAM
  - Keep bigger data structures on disk (databases) on DISK
- It is harder than you think… data usage evolves over a program's execution

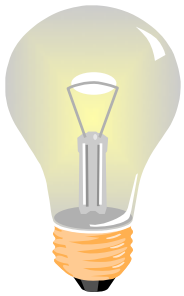
Best of Both Worlds

What we REALLY want: A BIG, FAST memory!
(Keep everything within instant access)

We'd like to have a memory system that
- PERFORMS like 2 GBytes of SRAM; but
- COSTS like 512 MBytes of slow memory.

SURPRISE: We can (nearly) get our wish!

KEY: Use a hierarchy of memory technologies:



Key IDEA

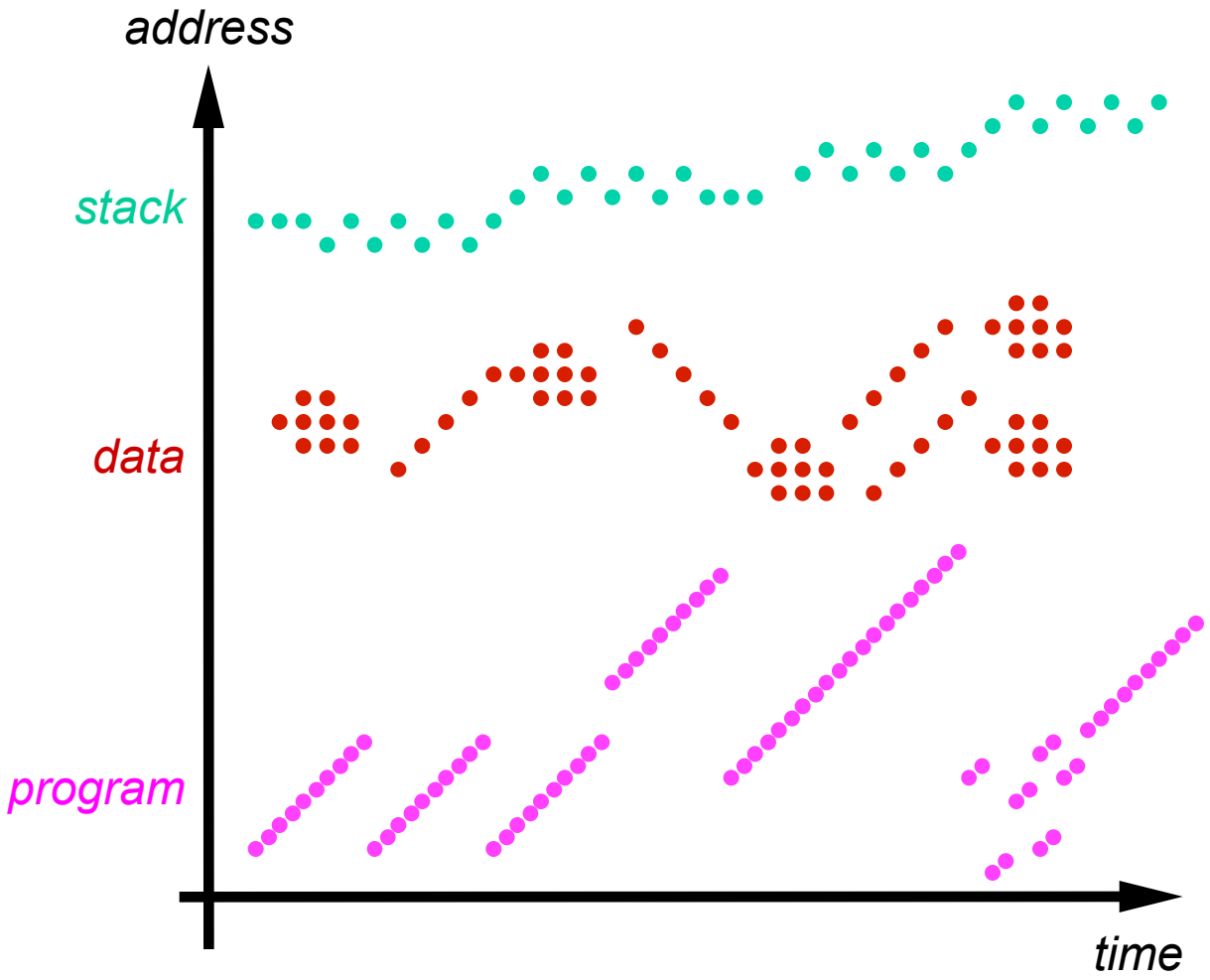
- Keep the most often-used data in a small, fast SRAM call a "Cache" ("on" CPU chip)
- Refer to Main Memory only rarely, for remaining data.

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.

Typical Memory Reference Patterns

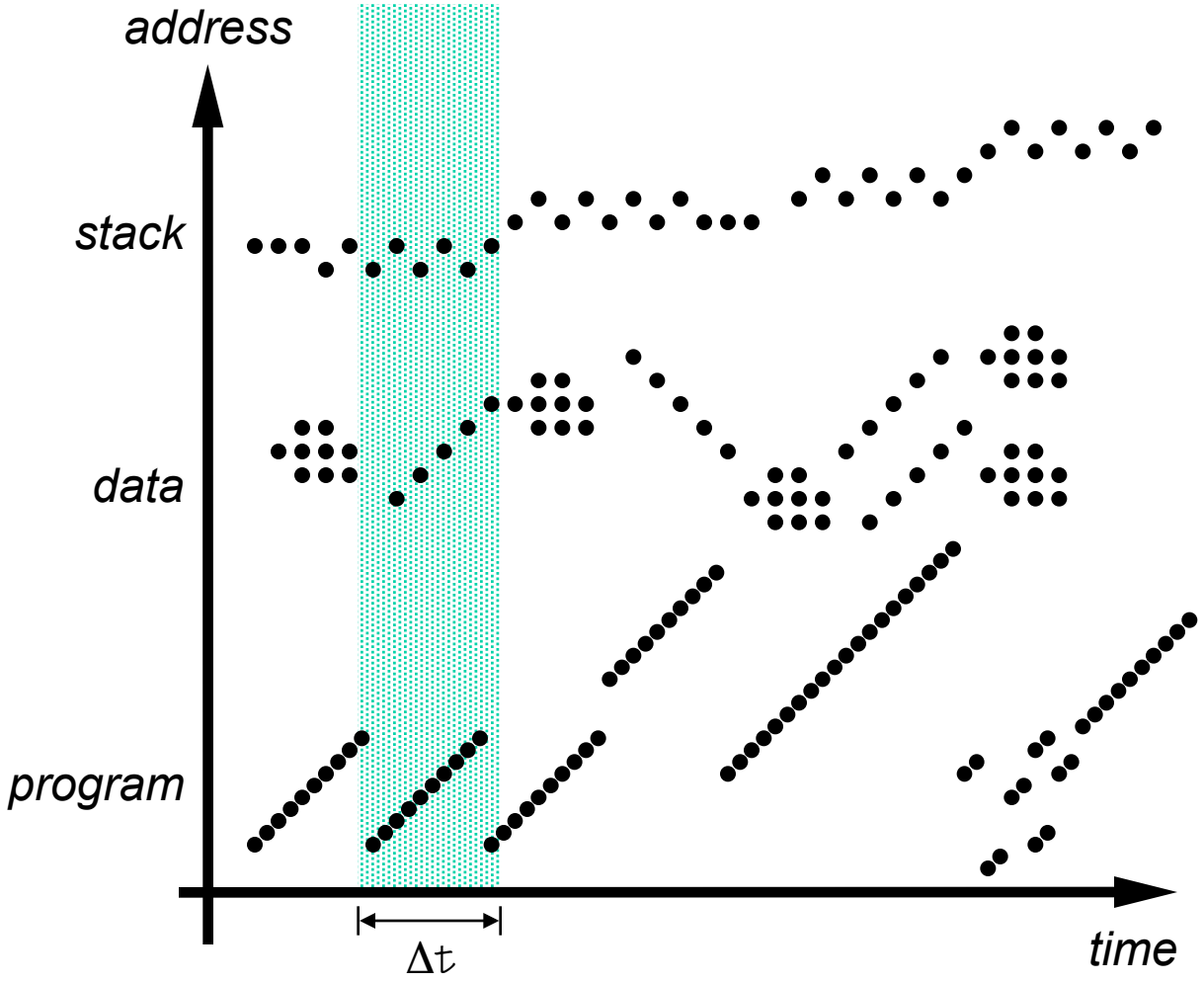


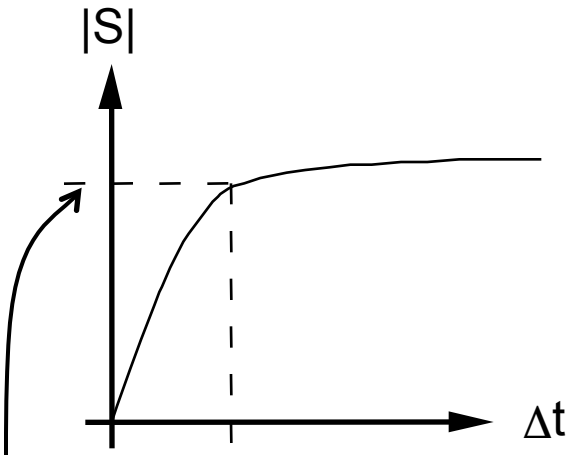
MEMORY TRACE –
A temporal sequence of memory references (addresses) from a real program.

TEMPORAL LOCALITY –
If an item is referenced, it will tend to be referenced again soon

SPATIAL LOCALITY –
If an item is referenced, nearby items will tend to be referenced soon.

Working Set





S is the set of locations accessed during Δt.

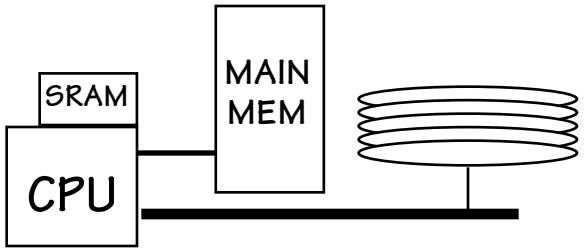
Working set: a set S which changes slowly w.r.t. access time.

Working set size, |S|

Exploiting the Memory Hierarchy

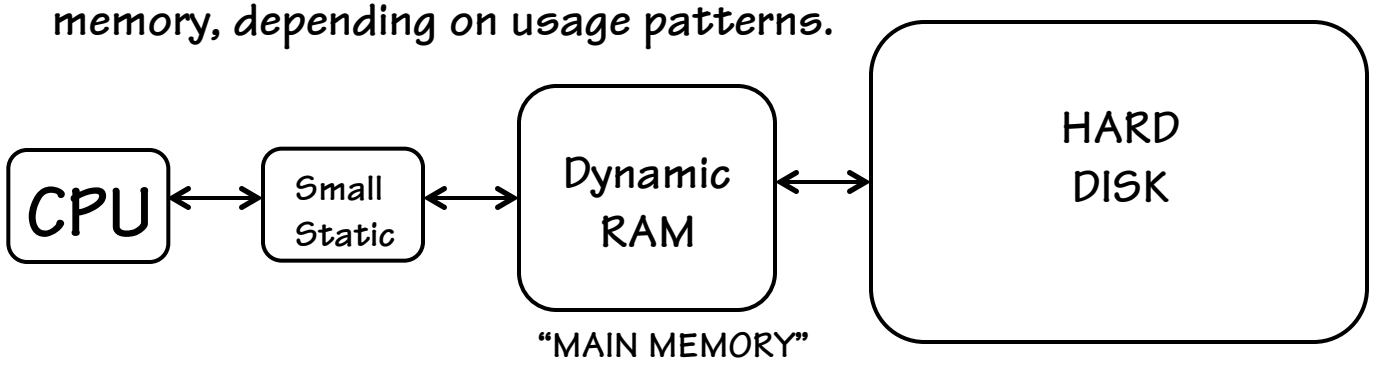
Approach 1 (Cray, others): Expose Hierarchy

- 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.

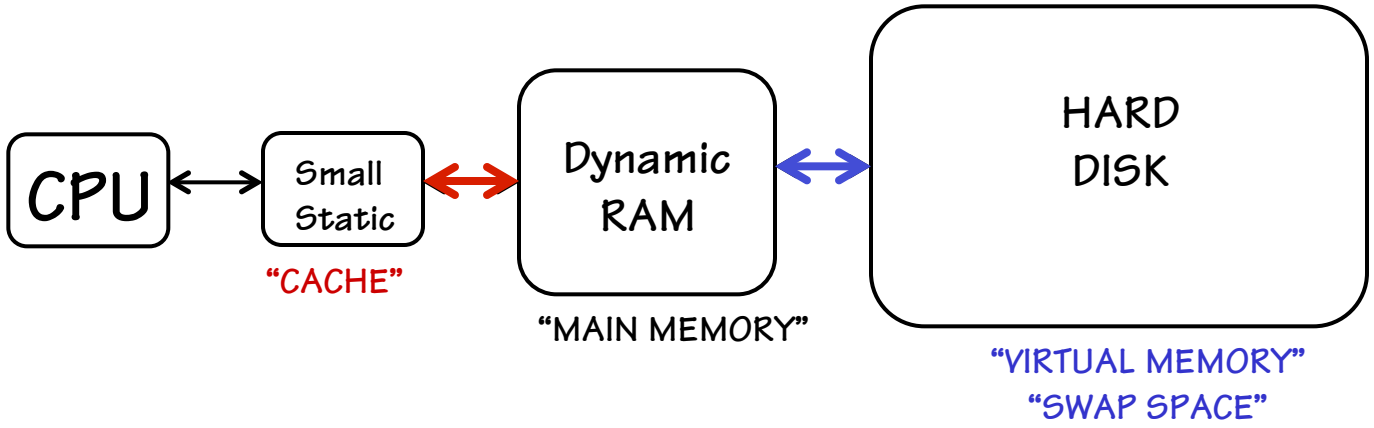


Why We Care

CPU performance is dominated by memory performance.

More significant than:

ISA, circuit optimization, pipelining, super-scalar, etc



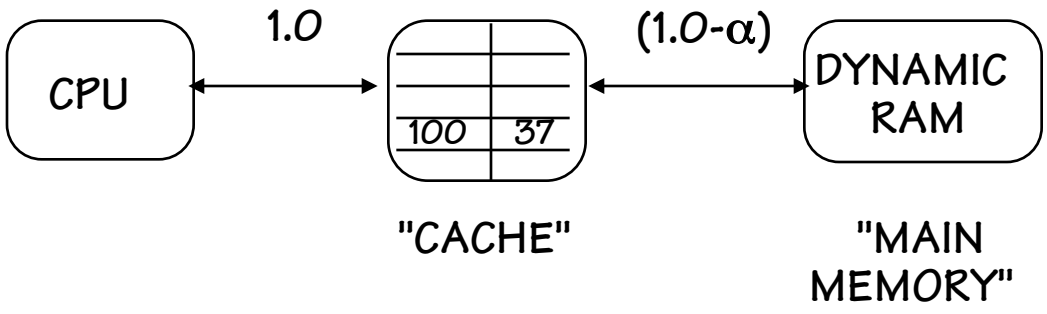
TRICK #1: How to make slow MAIN MEMORY appear faster than it is.

Technique: CACHEING – This and next Lectures

TRICK #2: How to make a small MAIN MEMORY appear bigger than it is.

Technique: VIRTUAL MEMORY – Lecture after that

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.

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

Challenge: To make the hit ratio as high as possible.

Why, on a miss, do I incur the access penalty for both main memory and cache?

2) Transparency (compatibility, programming ease)

How High of a Hit Ratio?

Suppose we can easily build an on-chip static memory with a 800 pS access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 10 nS. How high of a hit rate do we need to sustain an average access time of 1 nS?

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

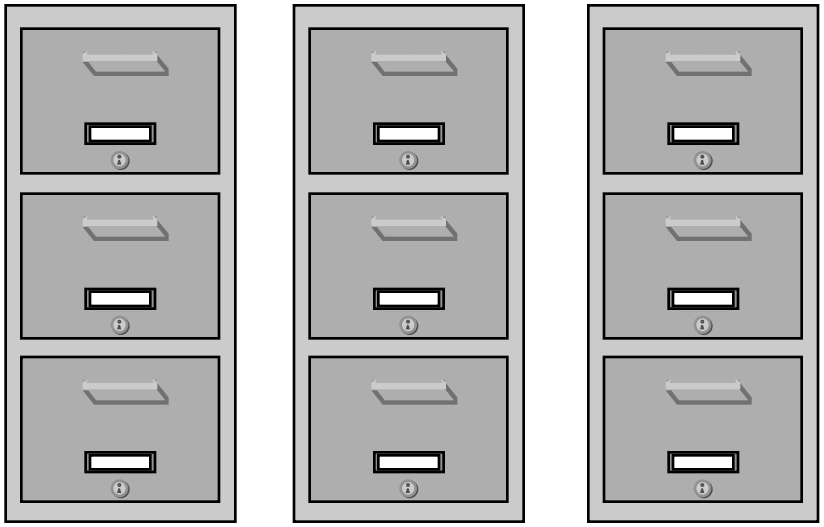
$$\alpha = 1 - \frac{t_{ave} - t_c}{t_m} = 1 - \frac{1 - 0.8}{10} = 98\%$$

WOW, a cache really needs to be good?

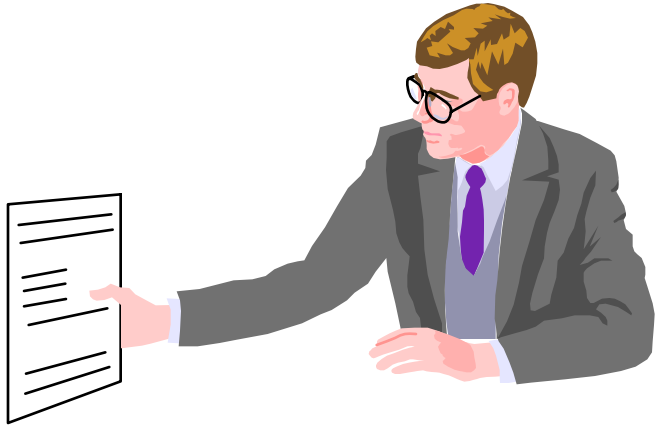
The Cache Principle

Find "Hart, Lee"

5-Minute Access Time:

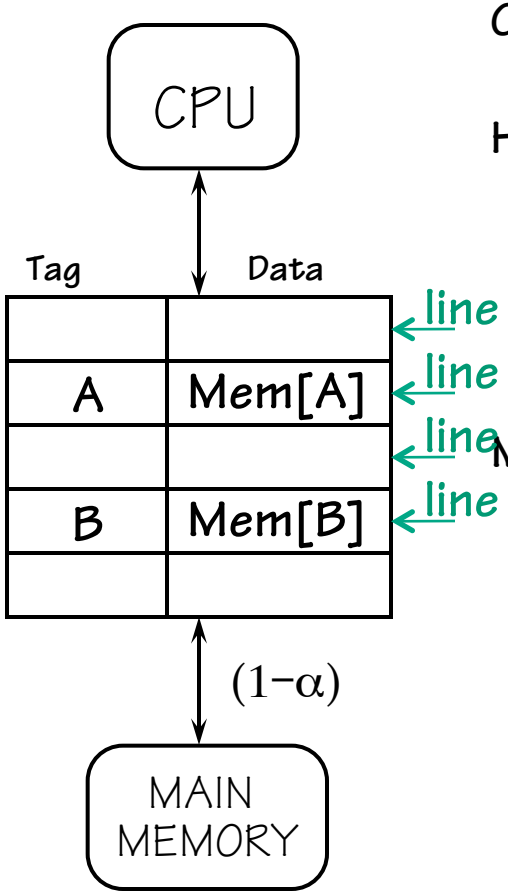


5-Second Access Time:



ALGORTHIM: Look on your desk 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]

Cache

Sits between CPU and main memory

Very fast memory that stores *TAGs* and *DATA*

TAG is the memory address (or part of it)

DATA is a copy of memory at the address given by TAG

Cache

	Tag	Data
Line 0	1000	17
Line 1	1040	1
Line 2	1032	97
Line 3	1008	11

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Access

On load we compare TAG entries to the ADDRESS we're loading

If Found → a *HIT*

return the DATA

If Not Found → a *MISS*

go to memory get the data

decide where it goes in the cache,

put it and its address (TAG) in the cache

Cache

	Tag	Data
Line 0	1000	17
Line 1	1040	1
Line 2	1032	97
Line 3	1008	11

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

How Many Words per Tag?

Caches usually get more data than requested (Why?)

Each *LINE* typically stores more than 1 word,
16-64 bytes (4-16 Words) per line is common

A bigger LINE SIZE means:

1) fewer misses because of spatial locality

2) fewer TAG bits per DATA bits

but bigger LINE means longer time on miss

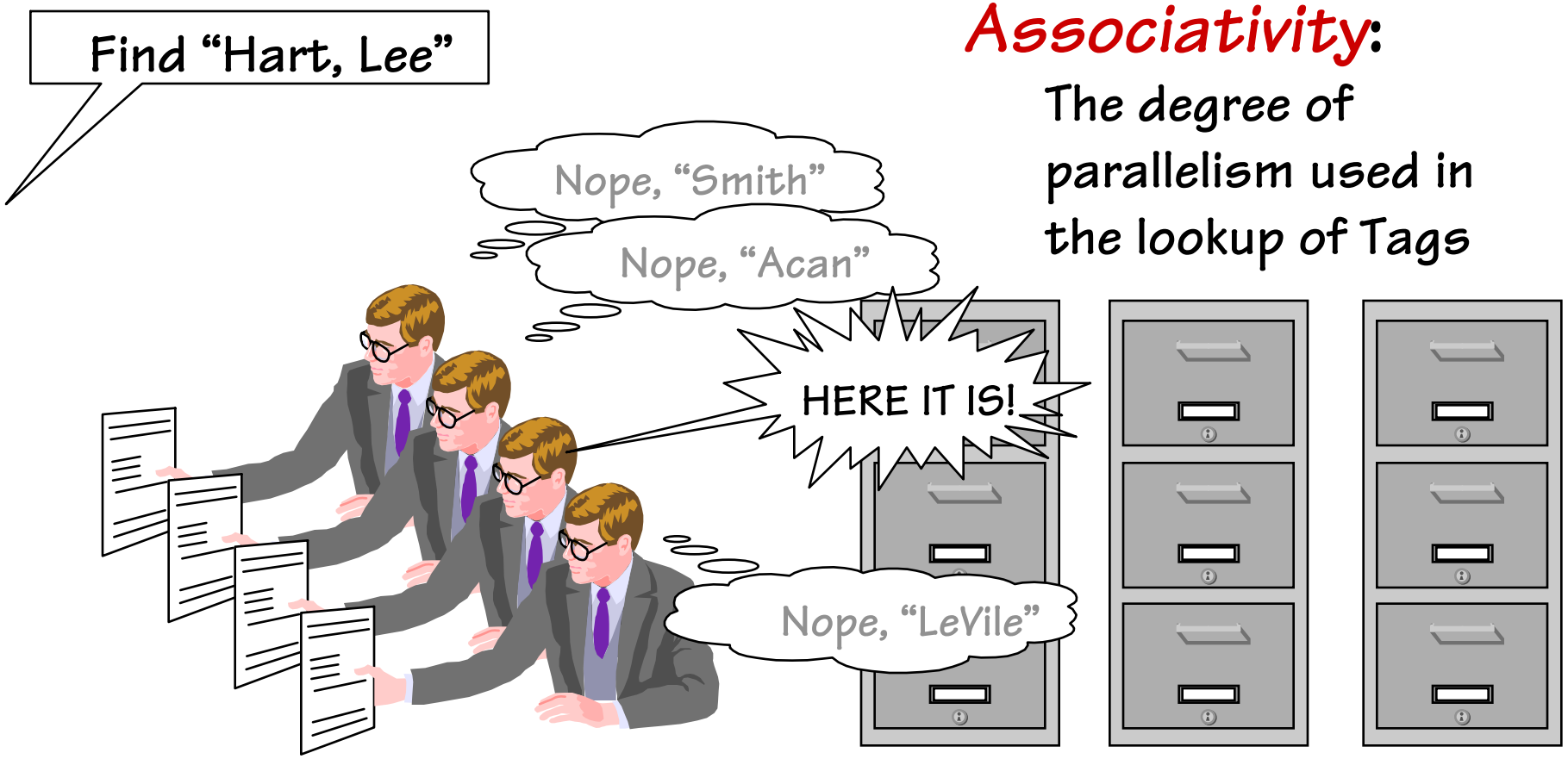
Cache

	Tag	Data	
Line 0	1000	17	23
Line 1	1040	1	4
Line 2	1032	97	25
Line 3	1008	11	5

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

How do we Search the Cache TAGs?



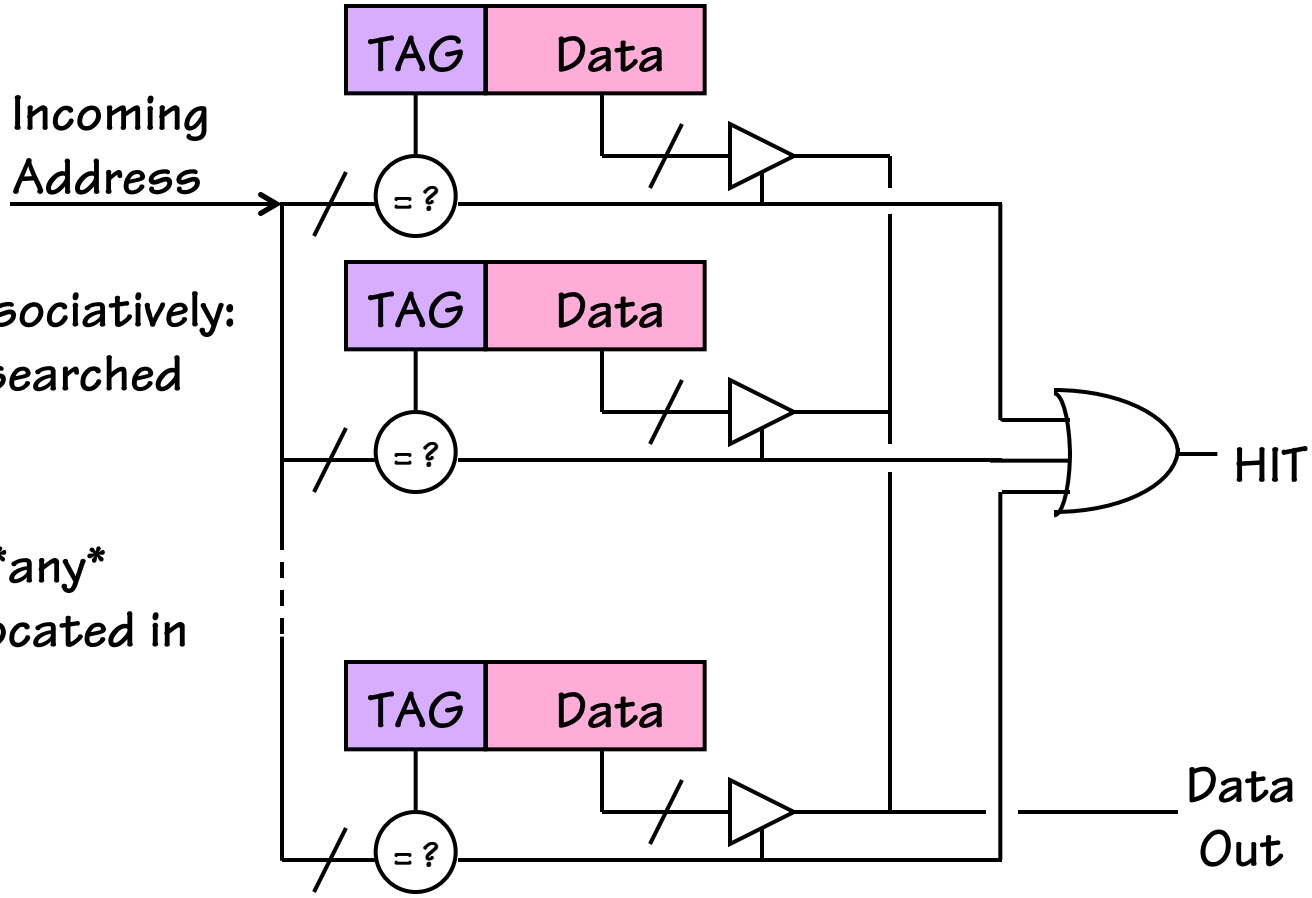
Associativity:

The degree of parallelism used in the lookup of Tags

Fully-Associative Cache

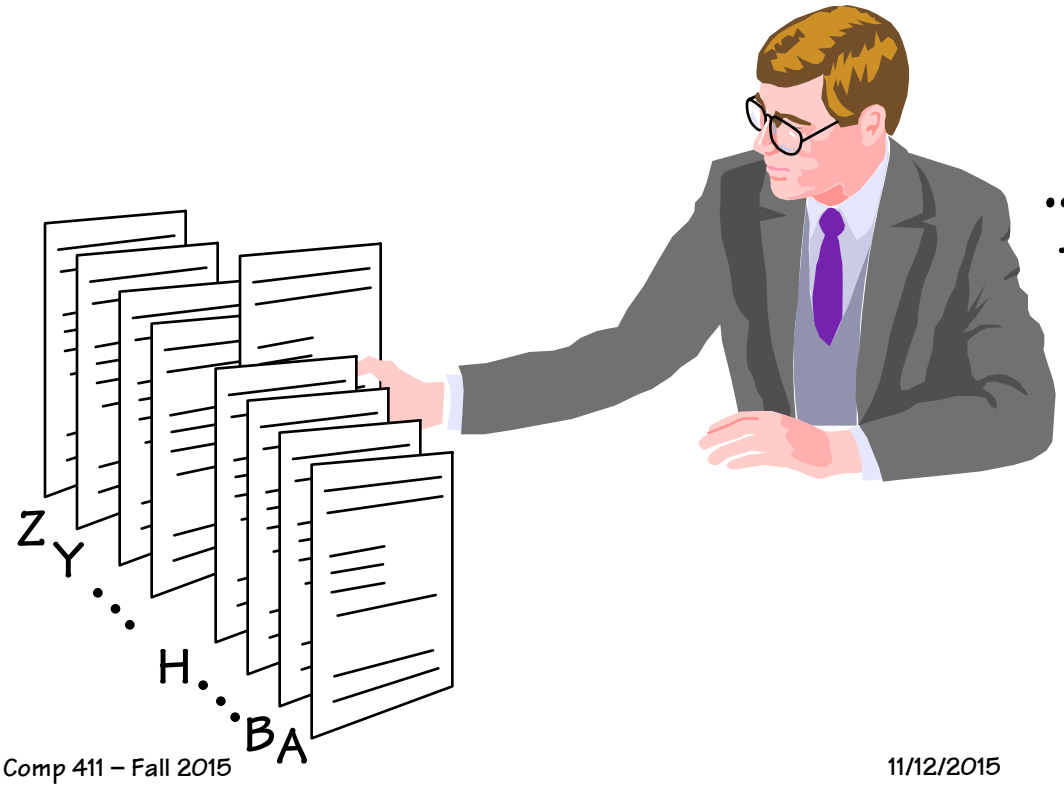
The extreme in associatively:
All TAGS are searched
in parallel

Data items from *any*
address can be located in
any cache line



Direct-Mapped Cache
(non-associative)

Find "Hart, Lee"



NO Parallelism:

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

... can use ordinary RAM as table

Direct-Map Example

With 8 byte lines, 3 low-order bits determine the byte within the line

With 4 cache lines, the next 2 bits determine which line to use

1024d = $1000000\mathbf{00}000_2$ → line = 00_2 = 0_{10}

1000d = $0111110\mathbf{1}000_2$ → line = 01_2 = 1_{10}

1040d = $1000001\mathbf{0}000_2$ → line = 10_2 = 2_{10}

Cache

	Tag	Data	
Line 0	1024	44	99
Line 1	1000	17	23
Line 2	1040	1	4
Line 3	1016	29	38

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Direct Mapping Miss

What happens when we now ask for address 1008?

$1008_{10} = 01111110000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

but earlier we put 1040 there...

$1040_{10} = 10000010000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

Cache

	Tag	Data	
Line 0	1024	44	99
Line 1	1000	17	23
Line 2	1008	11	5
Line 3	1016	29	38

Memory

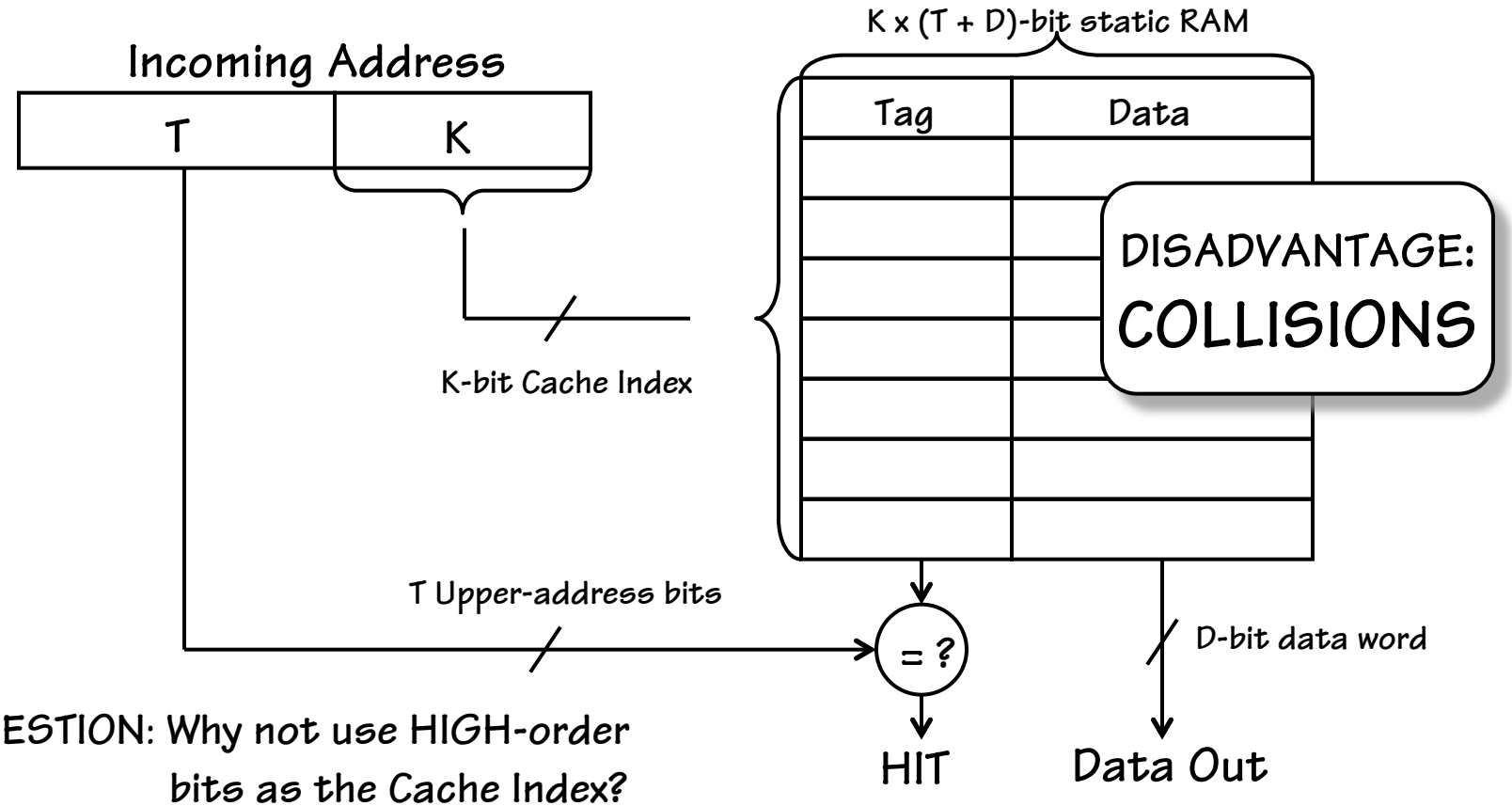
1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Direct Mapped Cache

LOW-COST Leader:

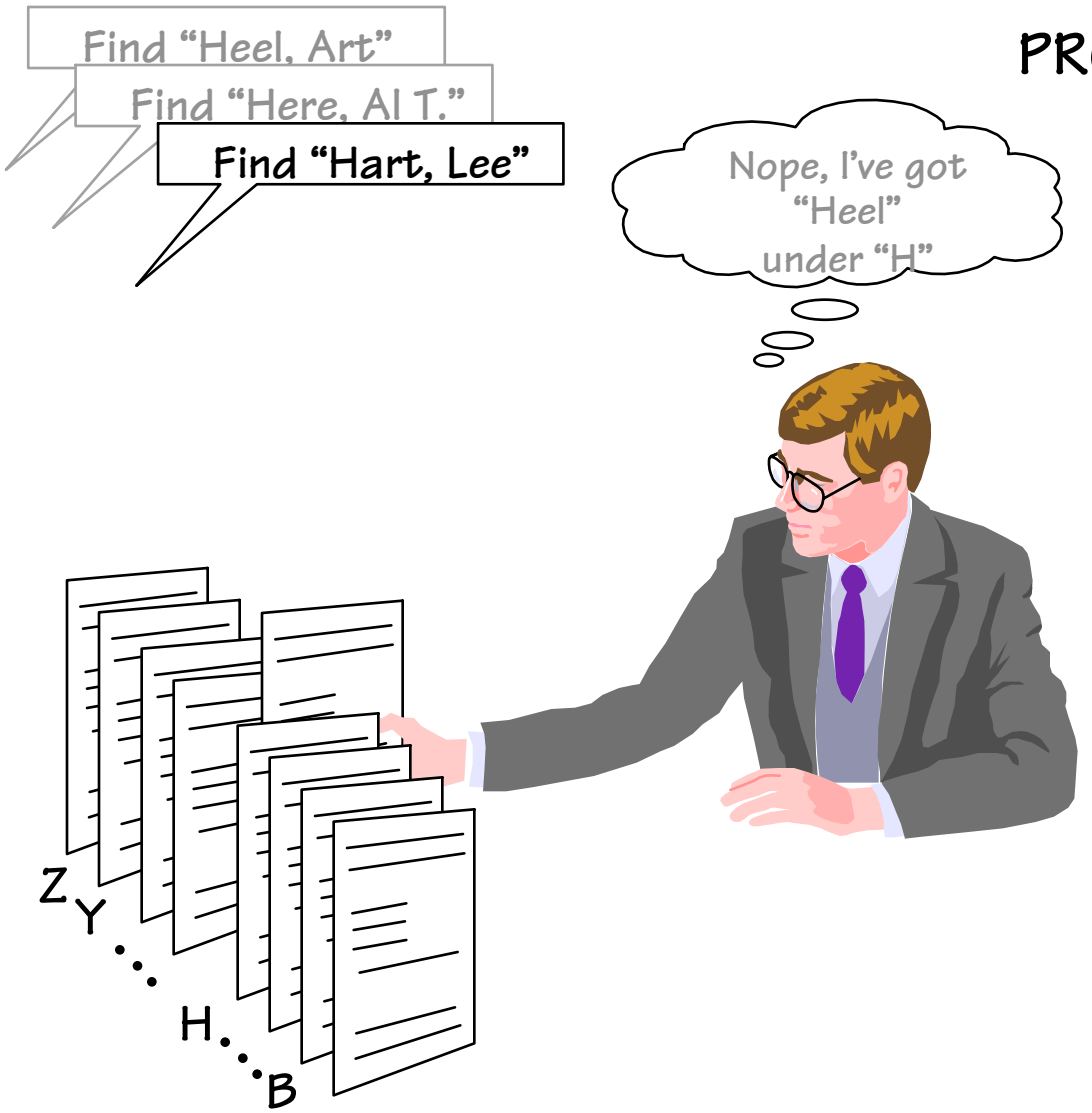
Requires only a single comparator and

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



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

A Problem with Collisions



PROBLEM:

Contention among H's....

- CAN'T cache both "Hart" & "Heel"

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

==> BETTER IDEA:

File by LAST letter!

Cache Questions = Cash Questions

What lies between Fully Associate and Direct-Mapped?

When I put something new into the cache, what data gets thrown out?

How many processor words should there be per tag?

When I write to cache, should I also write to memory?

What do I do when a write misses cache, should space in cache be allocated for the written address.

What if I have INPUT/OUTPUT devices located at certain memory addresses, do we cache them?

- Answers: Stay Tuned