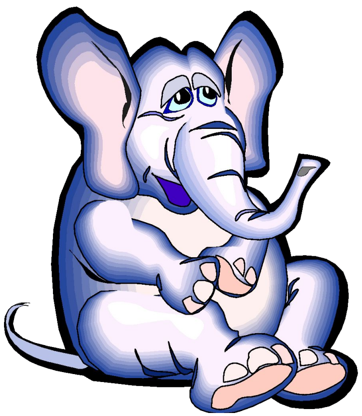
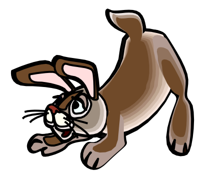
Memory Hierarchy & Caching

- Memory Flavors
- Principle of Locality
- Memory Hierarchies
- Caches
- Associativity
- Write-through
- Write-back

All Memories aren't created equal

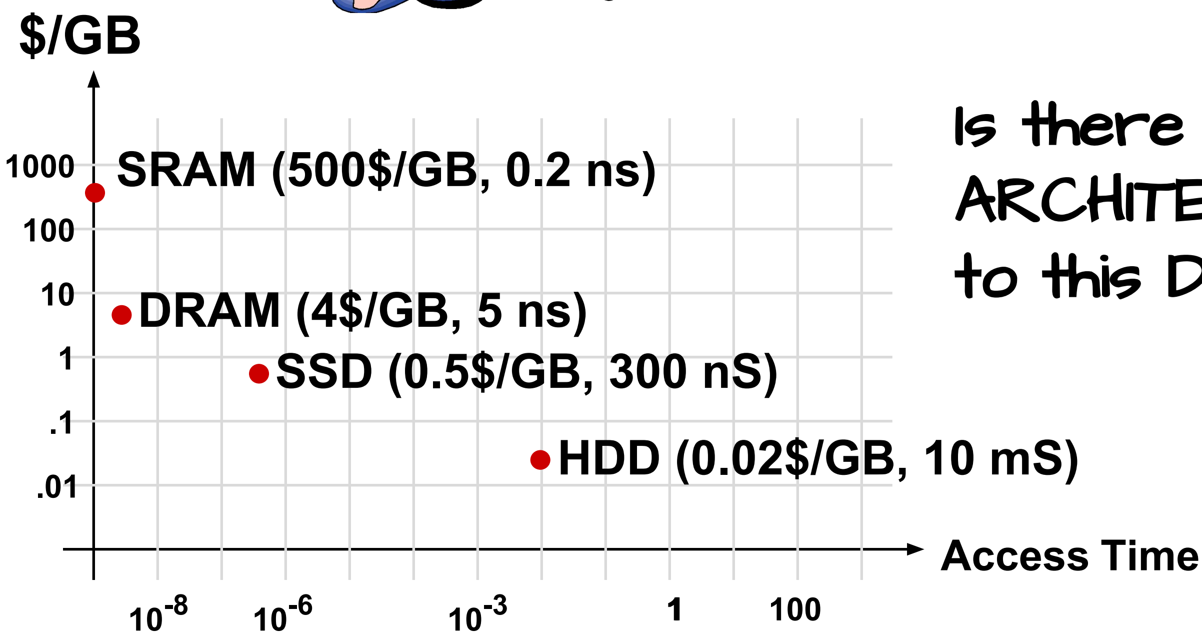




Quantity vs Speed…

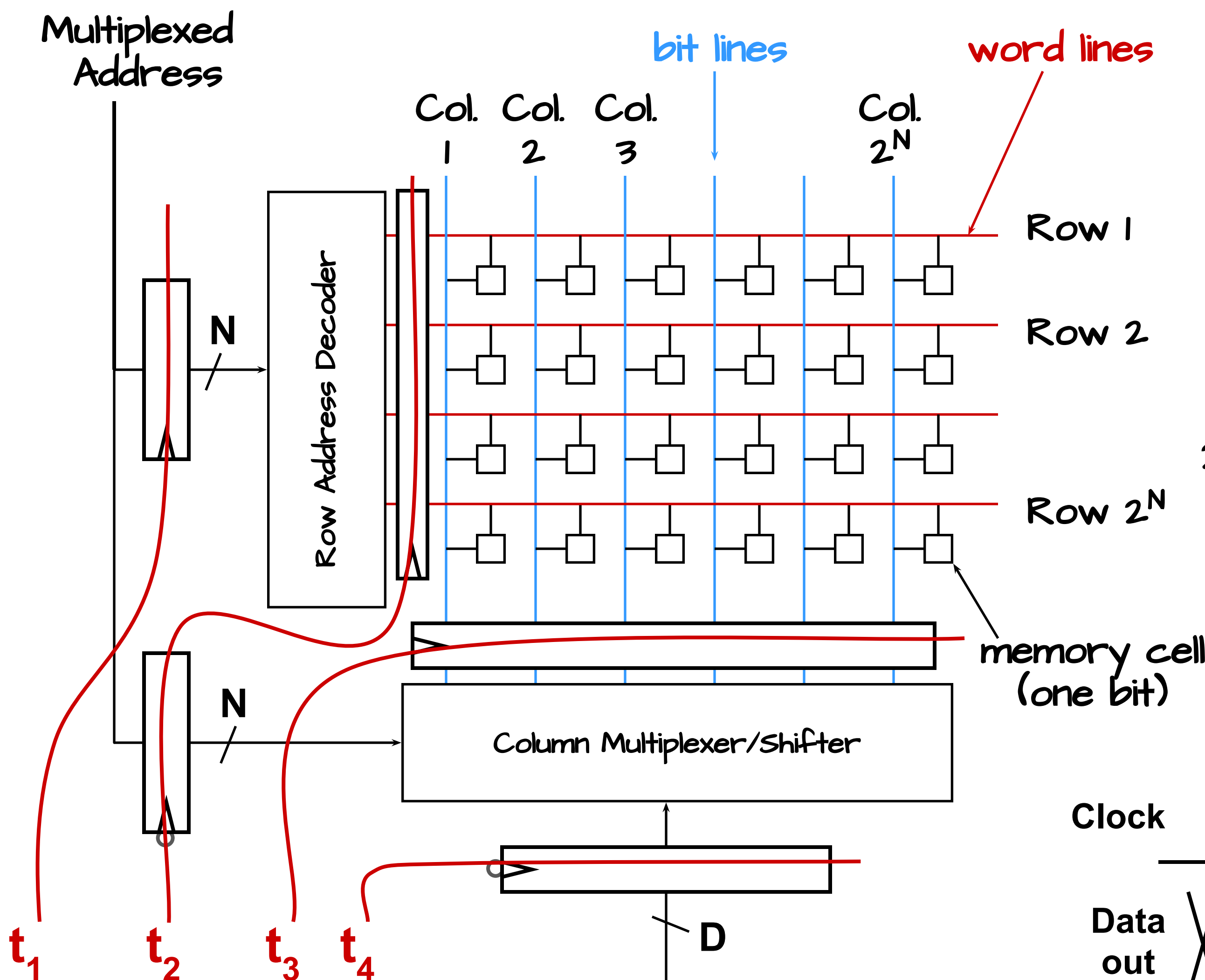
Memory systems can be either:
- BIG and SLOW…

  or
- SMALL and FAST.



Is there an ARCHITECTURAL solution to this DILEMMA?

Tricks for Increasing Throughput



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

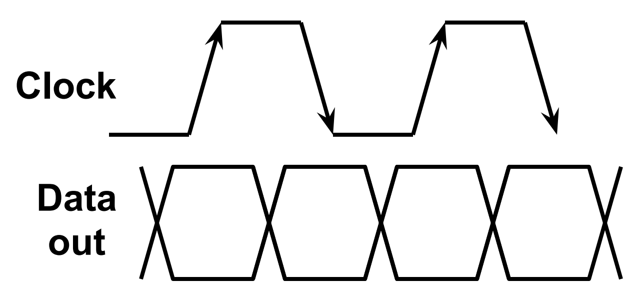
PIPELINING

Synchronous DRAM (SDRAM)

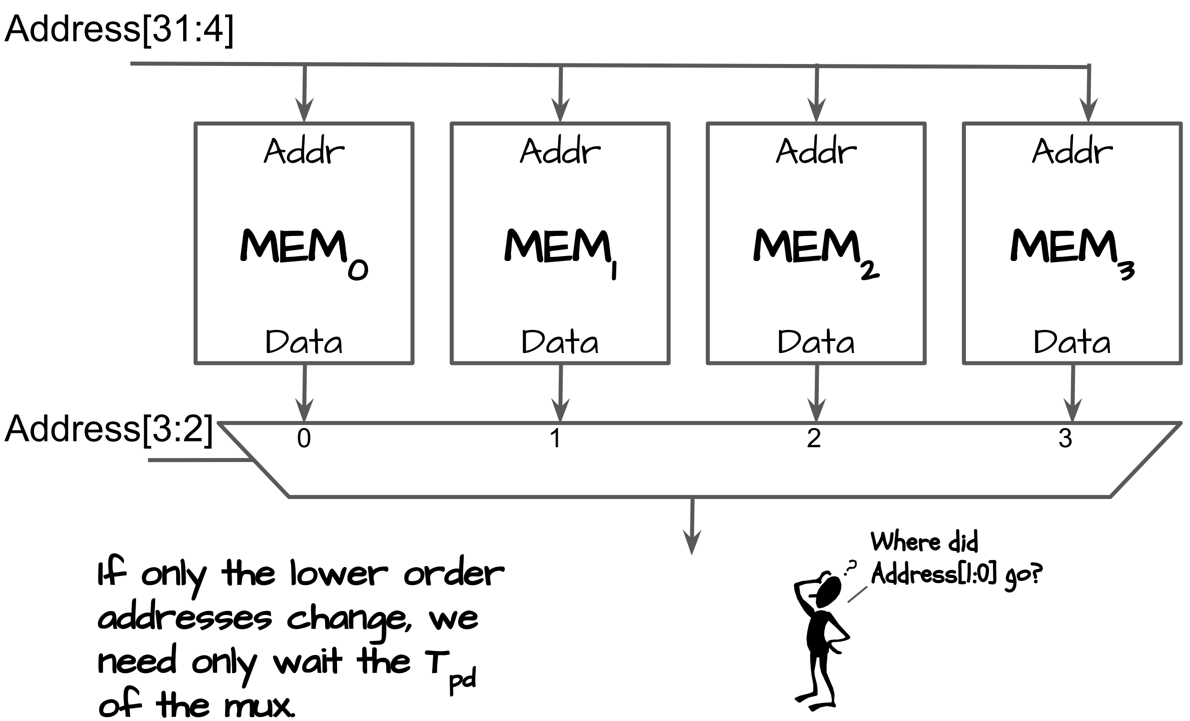
20ns reads and writes

($5 per Gbyte)

Double Data Rate Synchronous DRAM (DDR)



Another Trick



If only the lower order addresses change, we need only wait the T_{pd} of the mux.

The second thing that should try when asked to speed up a digital design...

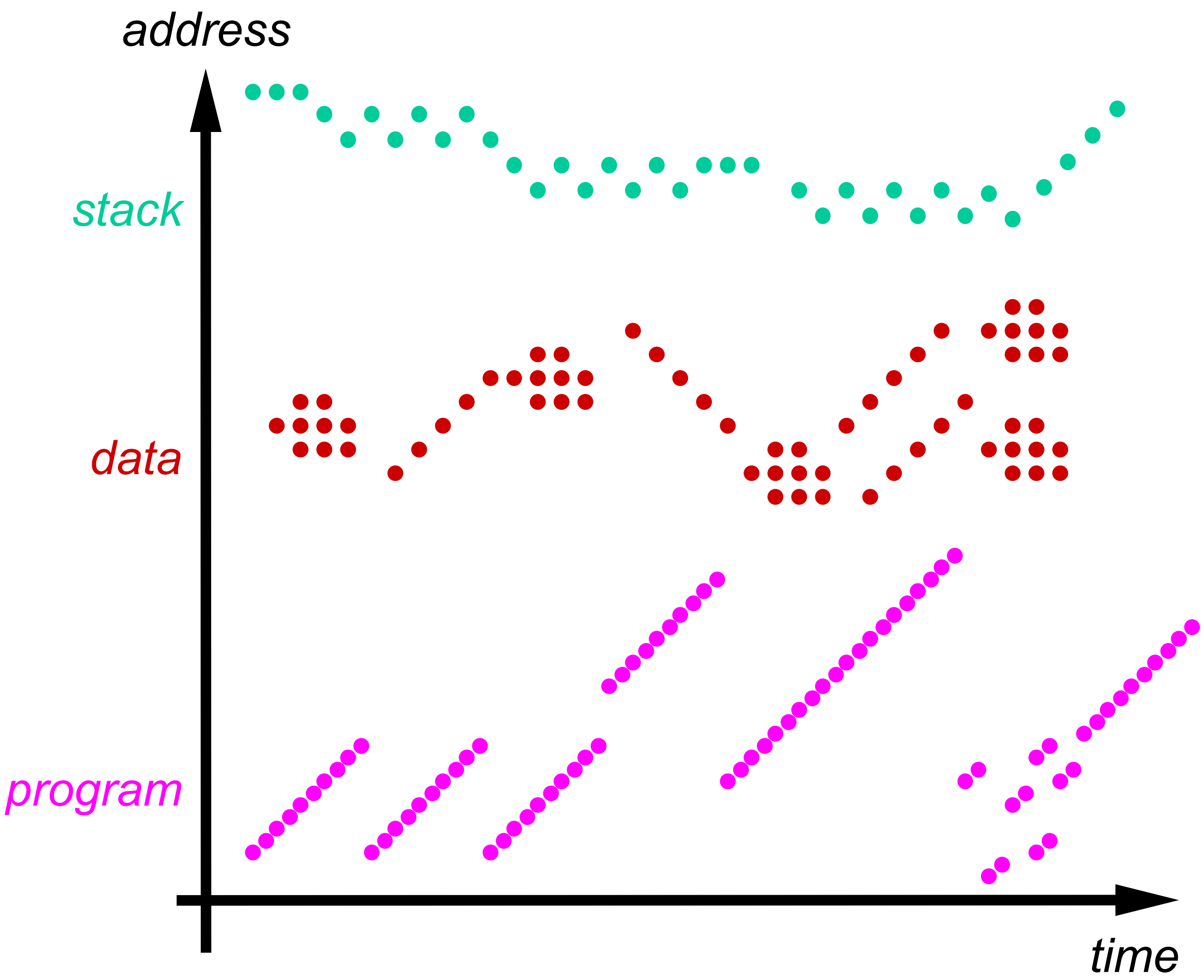
Interleaving

Accessing 4 memories at the same time has 4x the throughput. Also, every 4th word is in a different memory.

A limitation of both pipelining and interleaving is their assumption that addresses are sequential!

Which is *approximately* true!

Typical Memory Reference Patterns



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

TWO KEY OBSERVATIONS:

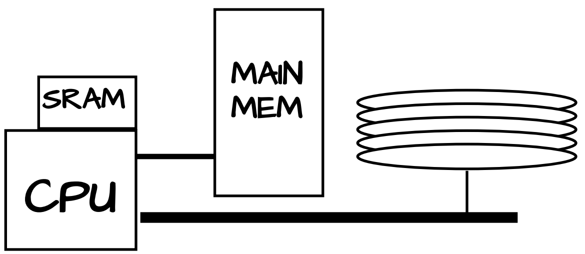
TEMPORAL LOCALITY -
If an item is referenced, it will tend to be referenced again soon

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

Exploiting the Memory Hierarchy

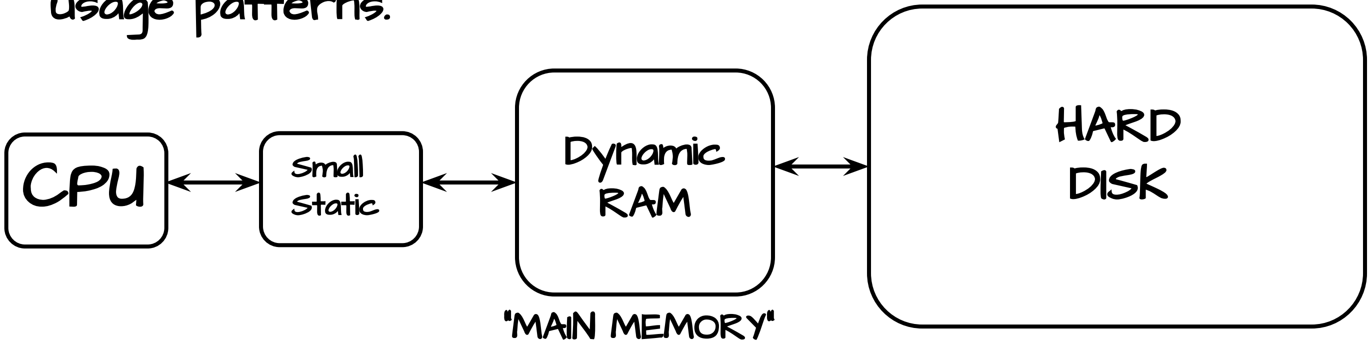
Approach 1 (Cray, others): Expose Hierarchy

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



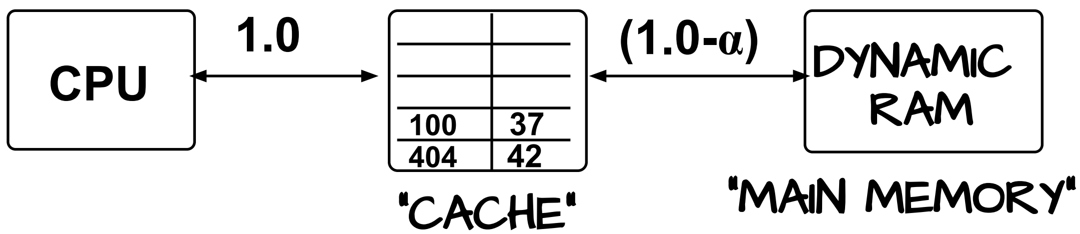
Approach 2: Hide Hierarchy

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



The Cache concept:

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, α, as high as possible.

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

How High of a Hit Ratio?

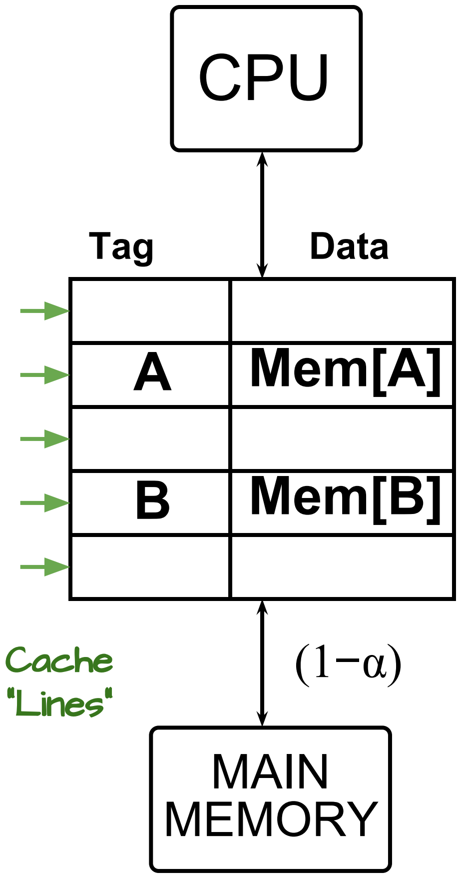
Suppose we can easily build an on-chip static memory with a 800 ps access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 10 ns. How high of a hit rate do we need to sustain an average access time of 1 ns?

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

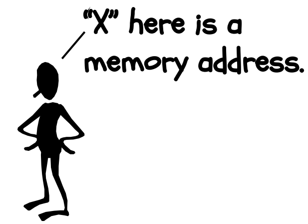
$$\alpha = 1 - (t_{ave} - t_c)/t_m = 1 - (1-0.8)/10 = 98\%$$

Wow, caches really need to be good! And they are!

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 any TAG of the cache

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

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

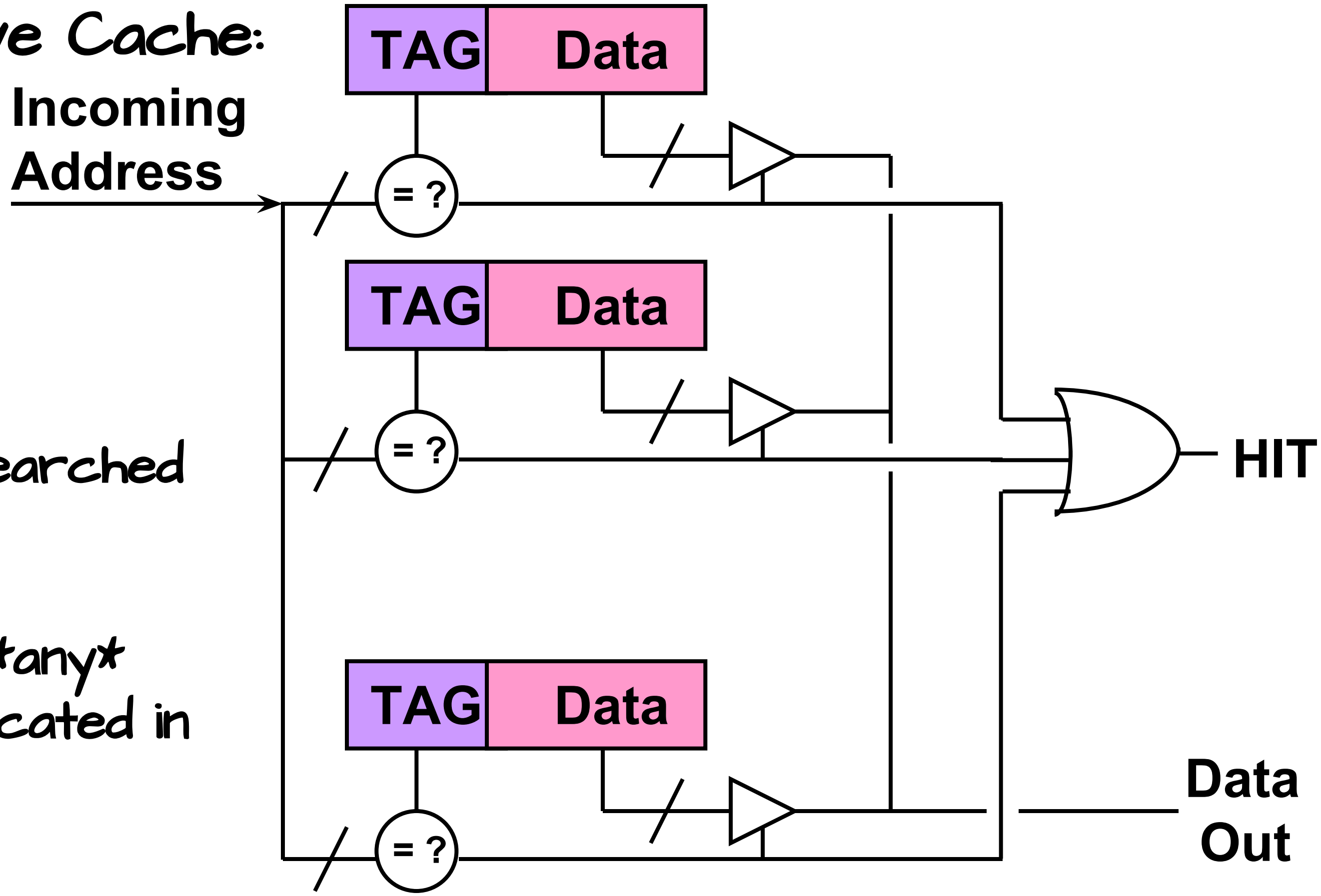
WRITE: Start Write to Mem(X)
Set TAG(k)=X, DATA(K)= new Mem[X]

Cache-lines might contain multiple sequential words from memory, thus amortizing the number of tag bits per data bits.

Searching for Tags

Associativity: Degree of parallelism used to lookup tags

Fully-Associative Cache:



The extreme in associatively:

All TAGS are searched in parallel

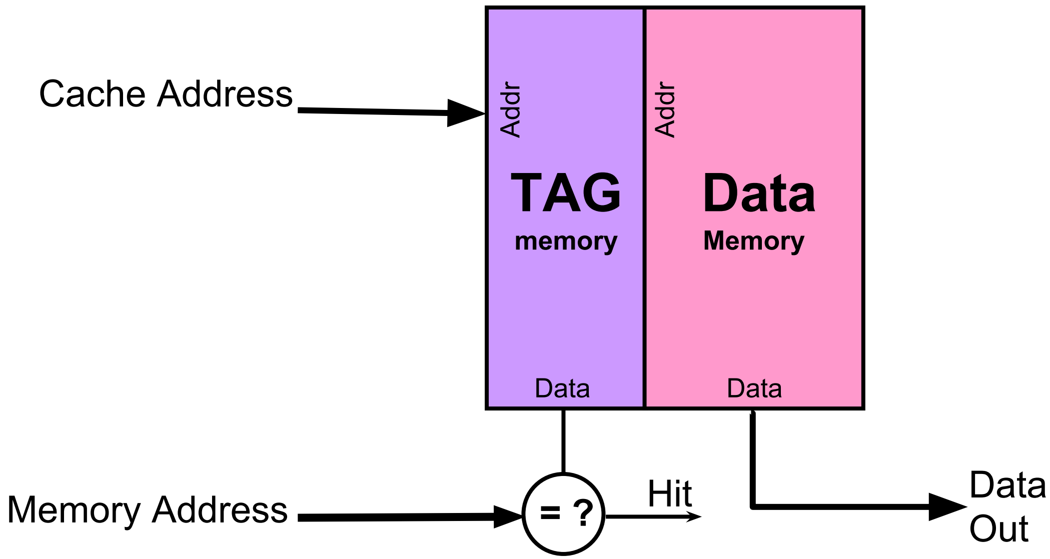
Data items from *any* address can be located in *any* cache line

The other extreme

Direct-mapped: If it is in cache it is in exactly one place

Non-associative or "one-way" associative. No parallelism.

Uses only one *comparator* and ordinary RAM for tags:



Low-cost leader:

Direct-mapped caches require a means for translating "Memory Addresses" to "Cache Addresses". A simple ***hash*** function.

Direct-Mapped Example

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

With 4 cache lines, the next 2 bits can be used to decide which line to use

$1024_{10} = 10000000000_2 \rightarrow \text{line} = 00_2 = 0_{10}$

$1000_{10} = 01111101000_2 \rightarrow \text{line} = 01_2 = 1_{10}$

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

Cache

	Tag	Data	
Line 0	1024	44	99
Line 1	1000	17	23
Line 2	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-Mapped Miss

What happens when we now ask for address 1008?

$1008_{10} = 0111111{\color{red}10}000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

but earlier we put 1040 there...

$1040_{10} = 100000{\color{red}10}000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

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

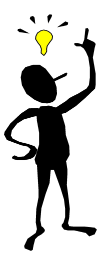
Fully-Assoc. vs. Direct-Mapped

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"
- Needs a replacement strategy to pick which line to use when loading new word(s) into cache

Direct-mapped N-line cache:

- One 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

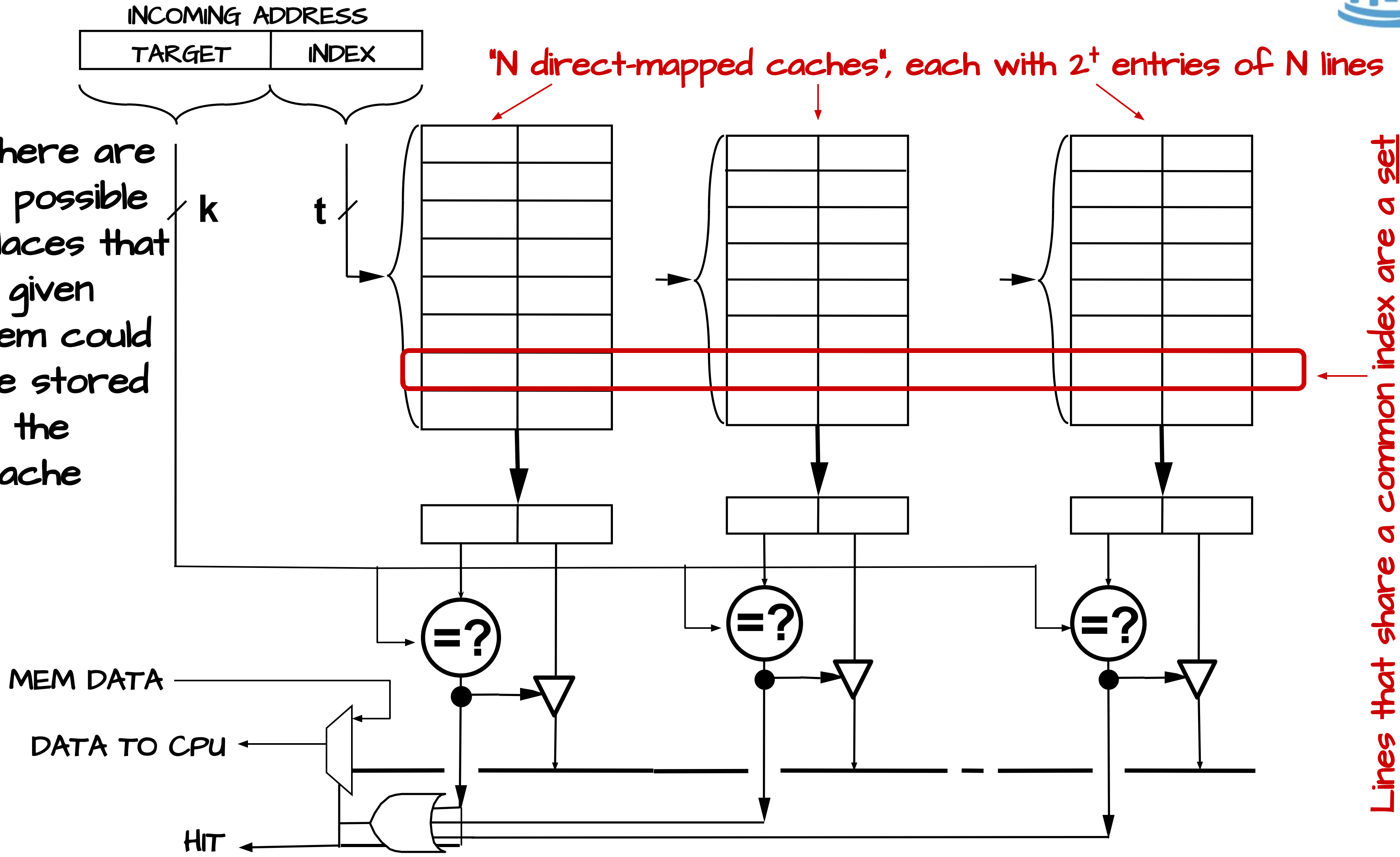


COLLISIONs occur when there are multiple items that we'd like to keep cached, we have room, but our management policies only keeps a subset of them.

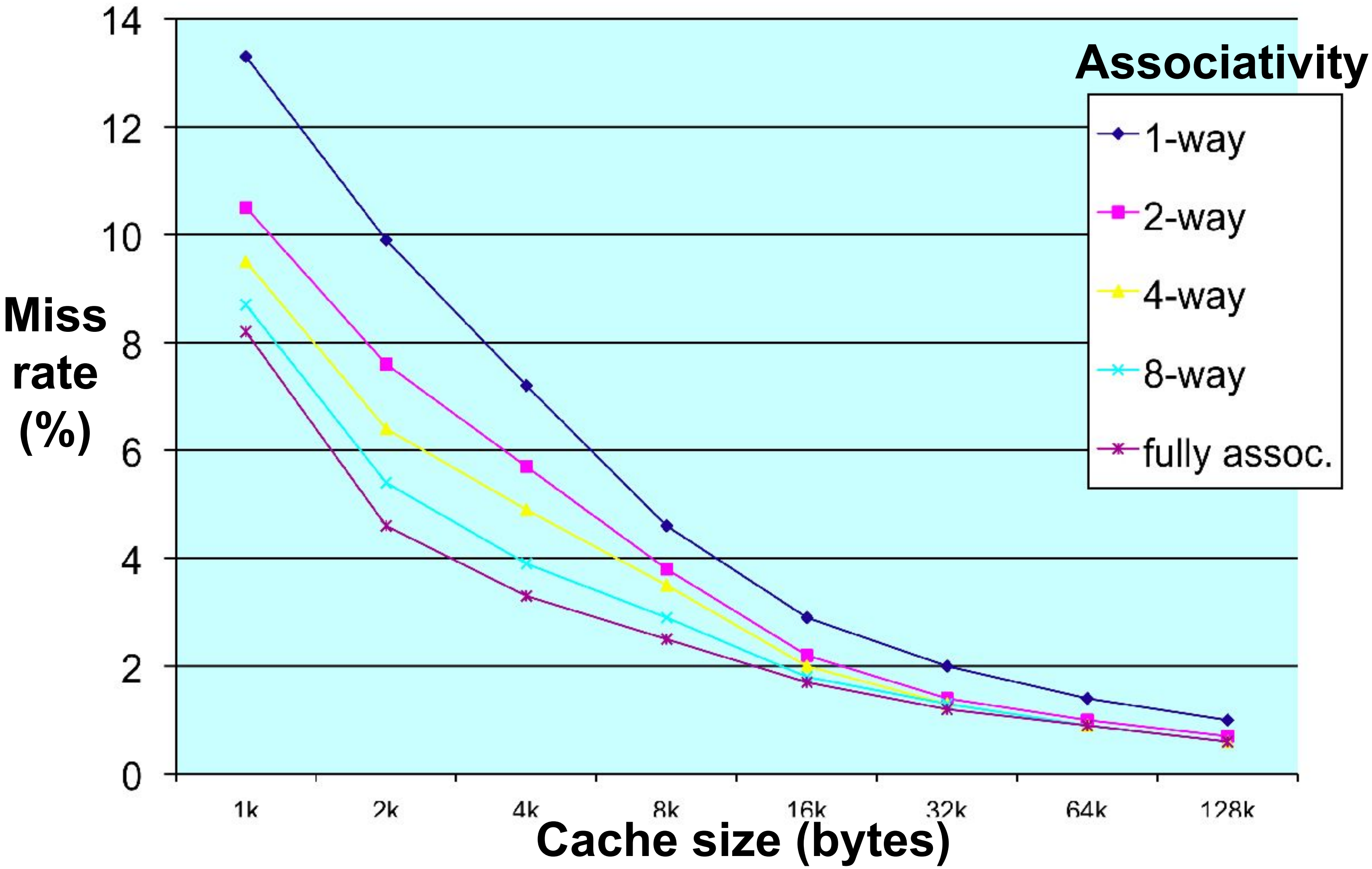
Is there something in-between?

N-Way Set-Associative Cache

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



Associativity vs. Miss Rate



8-way is (almost) as effective as fully-associative

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

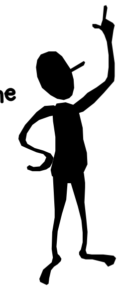
ON REFERENCE TO Mem[X]: Look for X among tags...

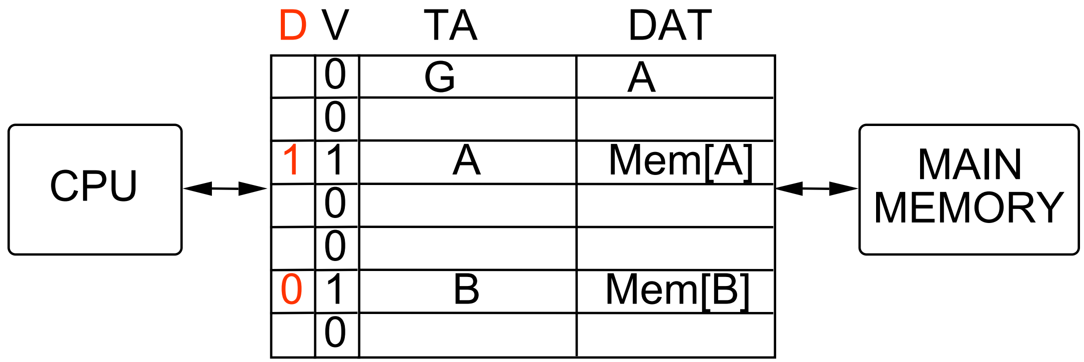
HIT: X = TAG(i) , for some cache line I
    READ: return DATA(i)
    WRITE: change DATA(i); ~~Start Write to Mem[X]~~

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]

Write-Back w/ "Dirty" bits

Dirty and Valid bits are per line not per set





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

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 the 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

Set TAG[k] = X, DATA[k] = new Mem[X], Read Mem[X]

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

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
- Write policies
- Write-through - Keeps memory and cache consistent, but high memory traffic
- Write-back - allows memory to become STALE, but reduces memory traffic

No simple answers, in the real-world cache designs are based on simulations using memory traces.