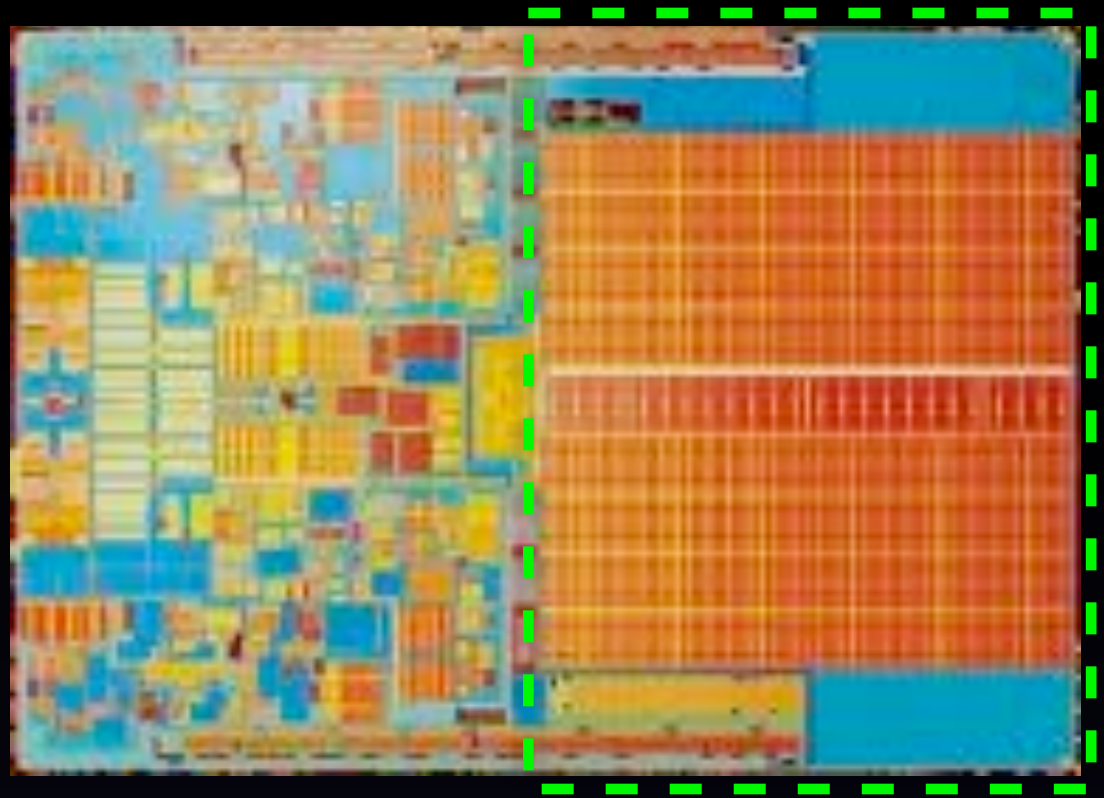
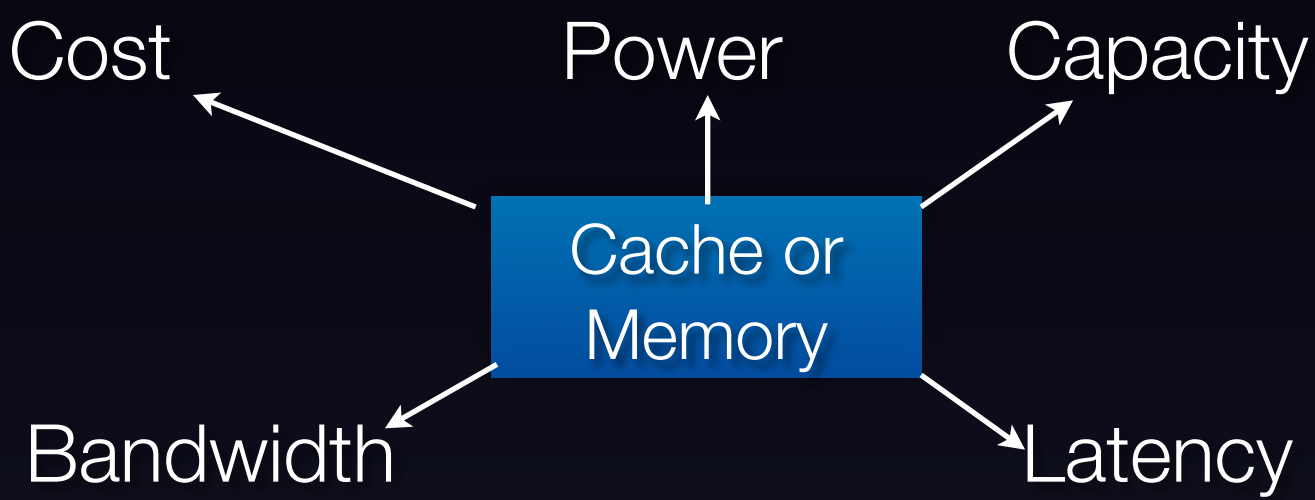


Lecture 2: Caches

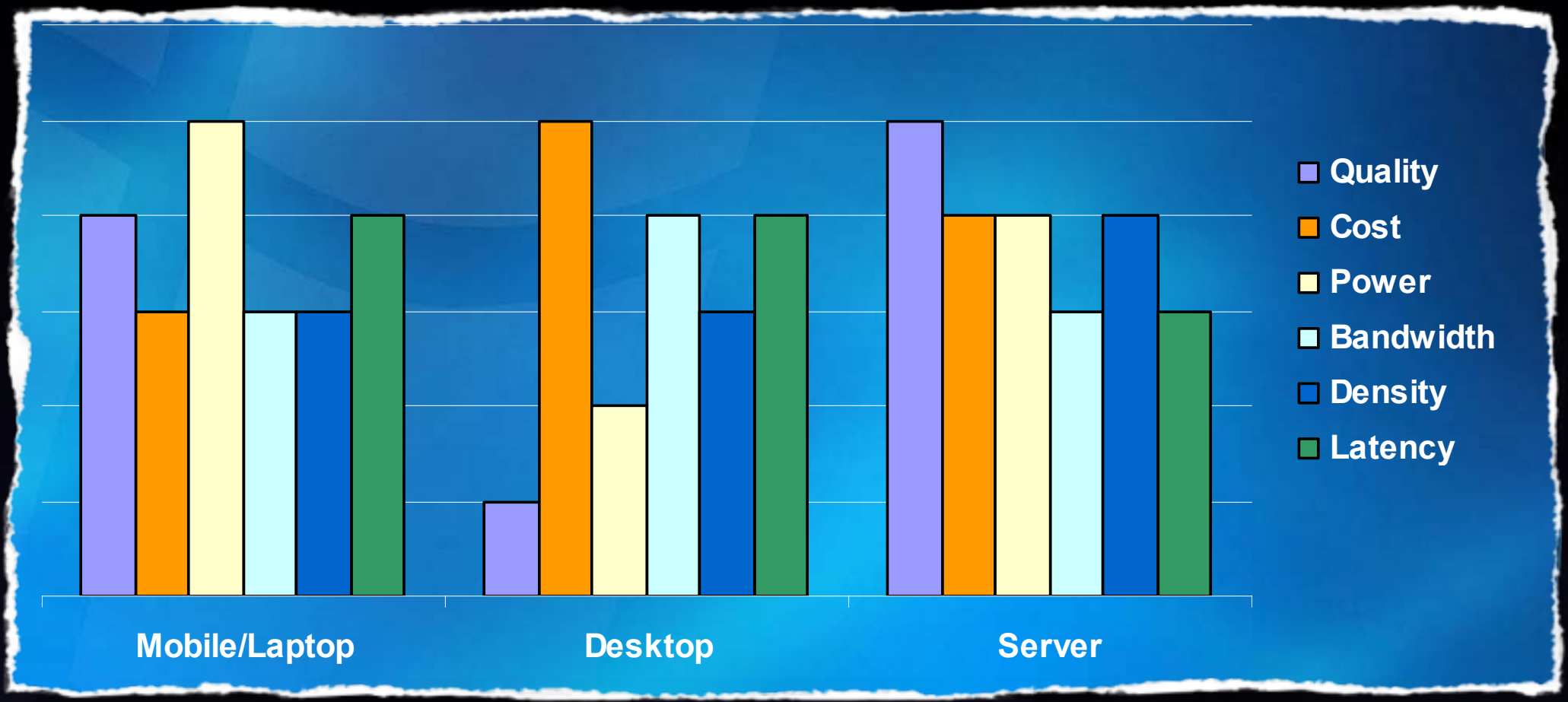
http://www.cs.sfu.ca/~ashriram/CS885/

Why focus on caches and memory ?

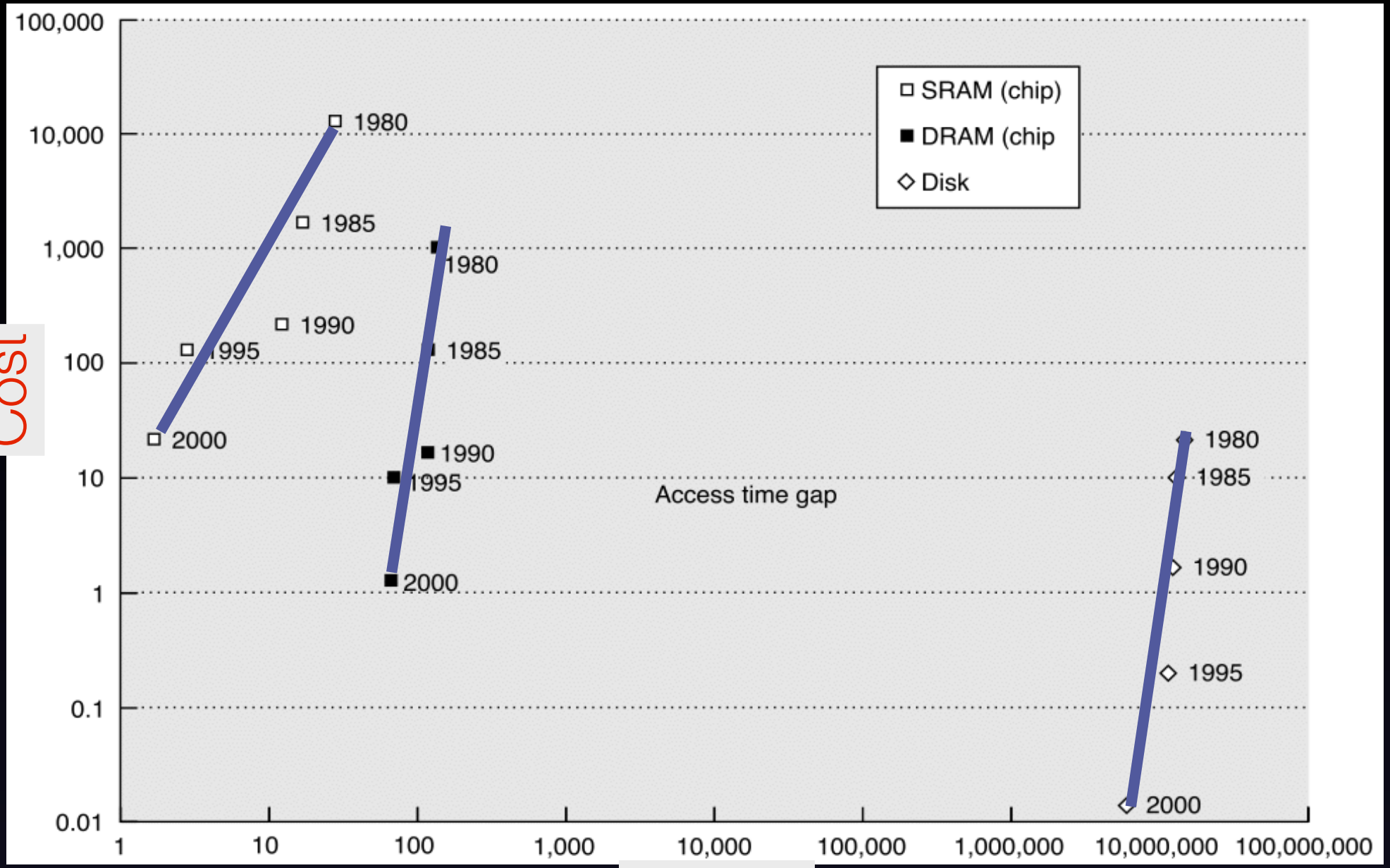
- CPU can only compute as fast as memory
  - Add operation takes 0.5ns; Memory is >100ns away
  - Data access dominates computing (Memory Wall)
- Occupies 2/3rds of total chip budget
- Design Goals



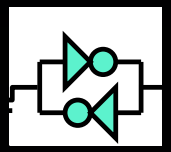
Market Forces



Memory Evolution

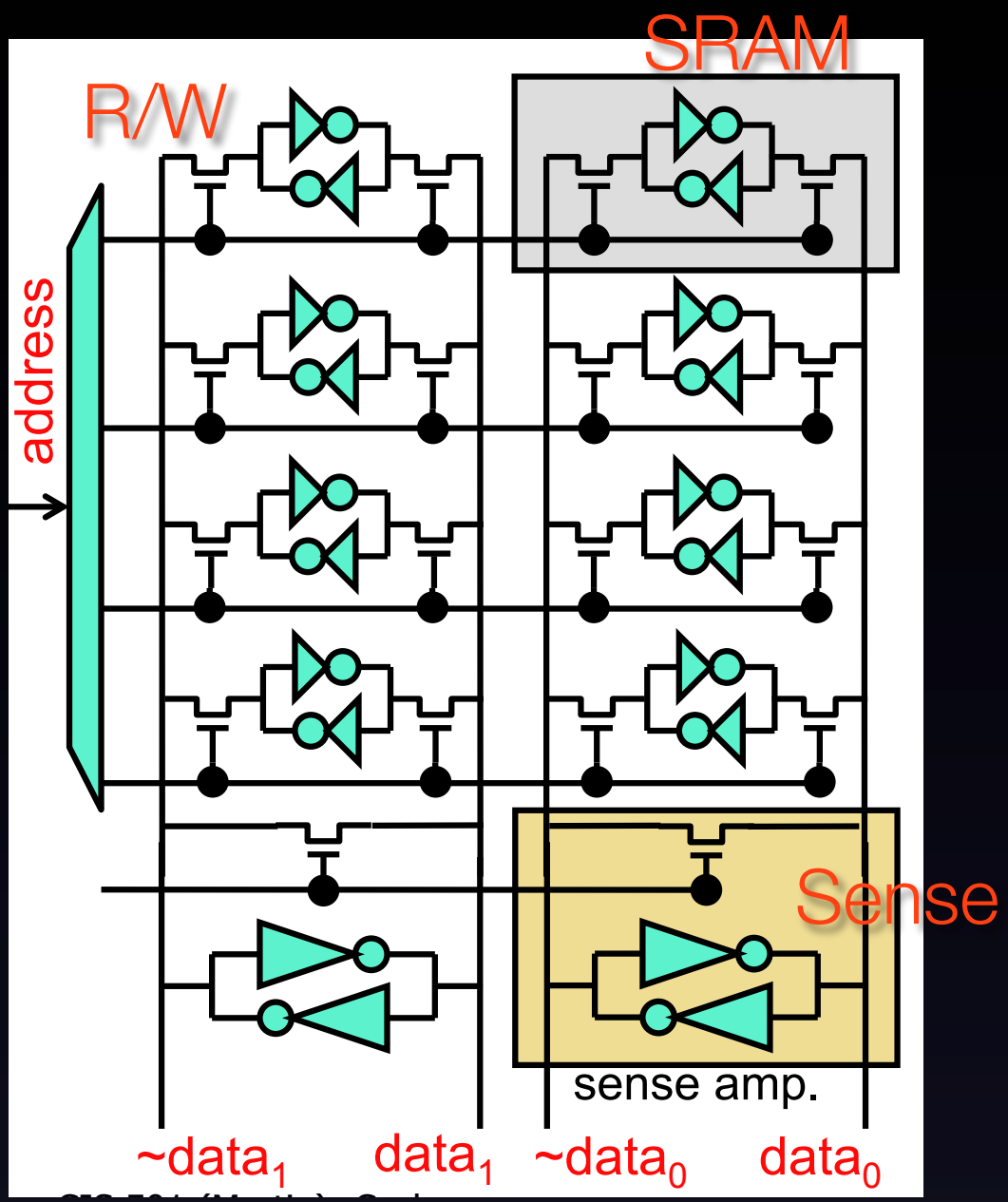


Types of Memory



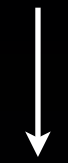
- SRAM
  - Uses same technology as CPUs
  - Essentially a logic loop
  - For Speed, Not Capacity
  - Access (sub ns); Speed proportional to capacity
- DRAM
  - Capacitative storage
  - Optimized for density and capacity
  - Slow (>40ns in the chip; 100ns to get to CPU)

Example SRAM



- SRAM ("Static RAM")
  - looped inverters hold state
- To read
  - equalize, swing, amplify
- To Write
  - overwhelm

Memory Technologies



- Cost (what can 200$ buy today 2010?)
  - SRAM 16MB
  - DRAM 4,000MB (4GB), 250x cheaper than SRAM
  - Flash 64,000 (64GB) - 16x cheaper than DRAM
  - Disk 2,000,000 (2TB) - 32x cheaper than Flash
- Latency
  - SRAM <1 to 2ns (on-chip)
  - DRAM ~50ns - 100x or more slower than SRAM
  - Flash 75,000 ns (75 µs) - 1500x vs DRAM
  - Disk 10,000,000 (10ms) - 133x vs FLash
- Bandwidth
  - SRAM 300 GB/s (12 port, 8 byte at 3 Ghz)
  - DRAM 25 GB/s ;
  - Flash 0.25 GB/s ; Disk 100MB/s

Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann
"Preliminary discussion of the logical design of an electronic computing instrument"
IAS memo 1946

Exploiting Locality

- Locality of memory references
  - interesting property of real programs; few exceptions
- Temporal Locality
  - recently referenced data likely to be used again
  - keep data in small and fast storage (**reactive**)
- Spatial Locality
  - Likely to access data near each other
  - fetch data in chunks (**Proactive**)

Library Analogy

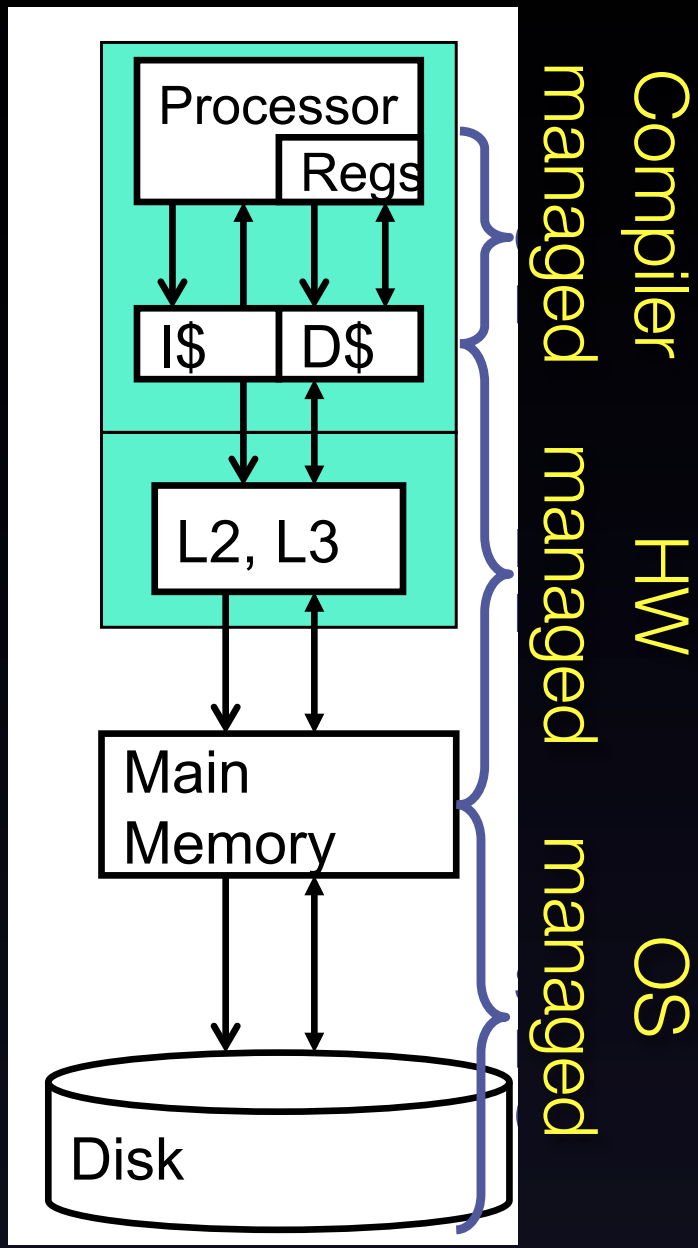
- Consider books in library
  - library has lots of books, but slow
  - far away (time to walk to library)
  - big (time to walk within library)

- How can you avoid latencies
  - check out books and put them on desk (limited capacity)
  - keep recently used books around (**Temporal locality**)
  - keep books on related topic together (**Spatial locality)**
  - Guess what books will be needed in the future (prefetching)

Memory Hierarchy: Exploiting Locality

- Hierarchy of memory components
  - Upper components; Fast, Small, expensive
  - Lower components; Slow, Big, Cheap
- Most frequently sed data in M1
  - move data up and down the hierarchy
- Optimize
  - Avg. Latency = $Latency_{hit}$ + %miss * $Latency_{miss}$

Memory Hierarchy

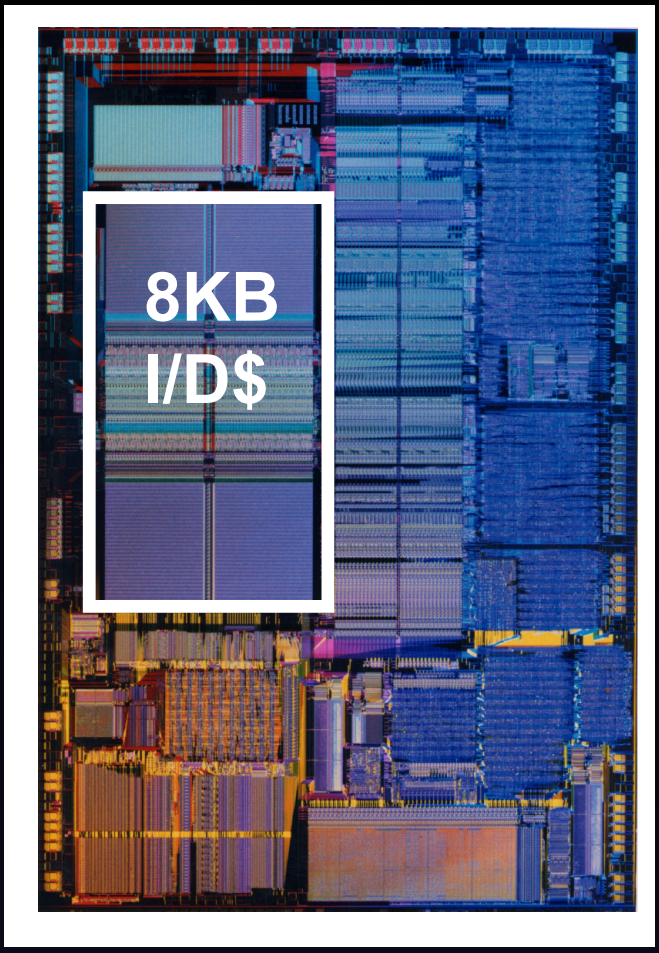


- Level 0 : Registers
- Level 1 : Split Ins. and Data cache
  - typically 8-64KB
  - inside core
- Level 2 and 3 (SRAM)
  - shared by cores
  - 2nd level typically 256-512KB
  - last-level (LLC) typicall 4-16MB
- Level 4 : Main Memory DRAM
  - Desk (4GB), Servers (100s GB)

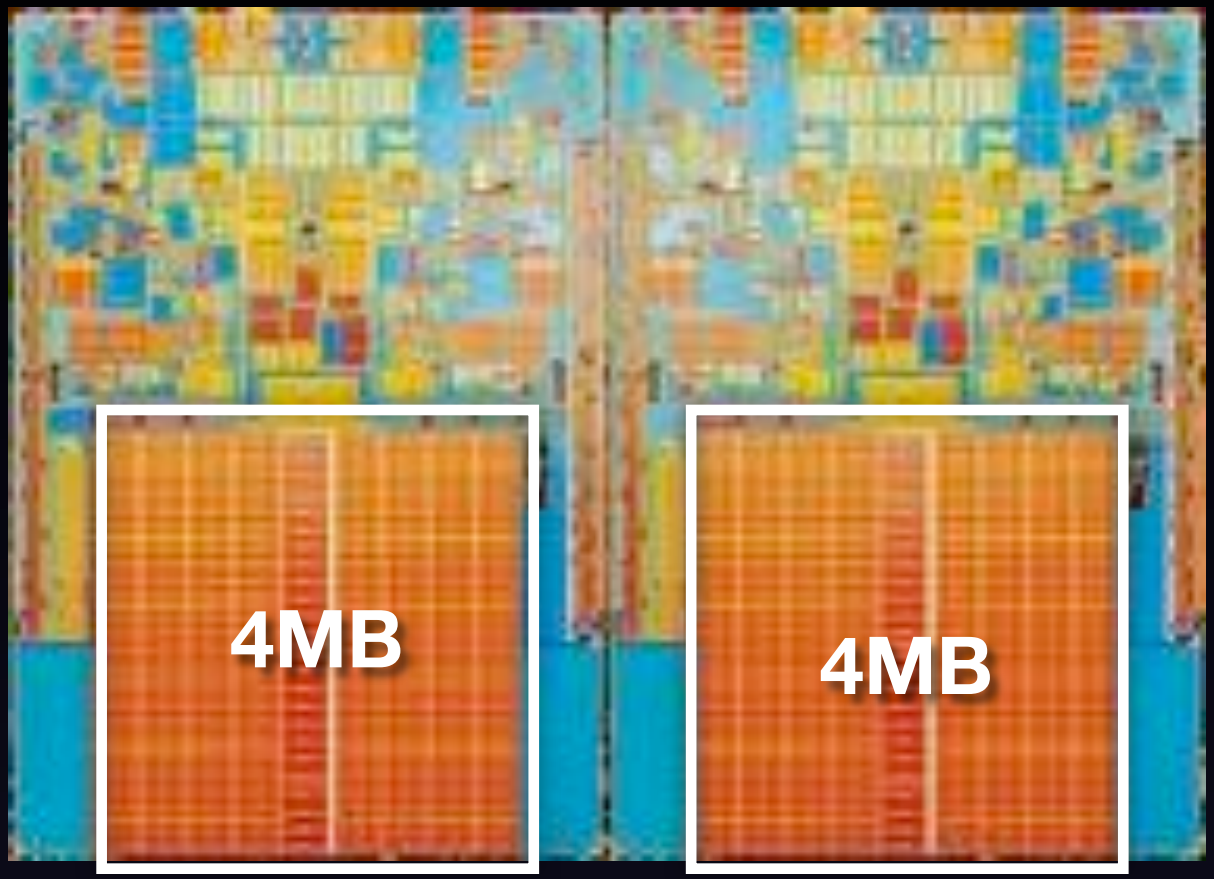
Library Analogy

- Registers = Books on desk
  - actively used, small capacity
- Caches = bookshelves
  - moderate capacity, pretty fast to access
- Main Memory = Library
  - Big; holds almost all data; but slow

Intel 486

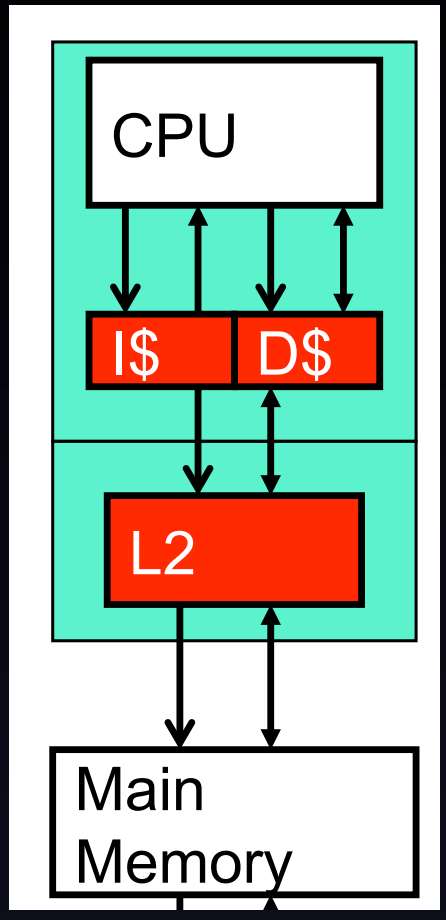


Intel Penryn



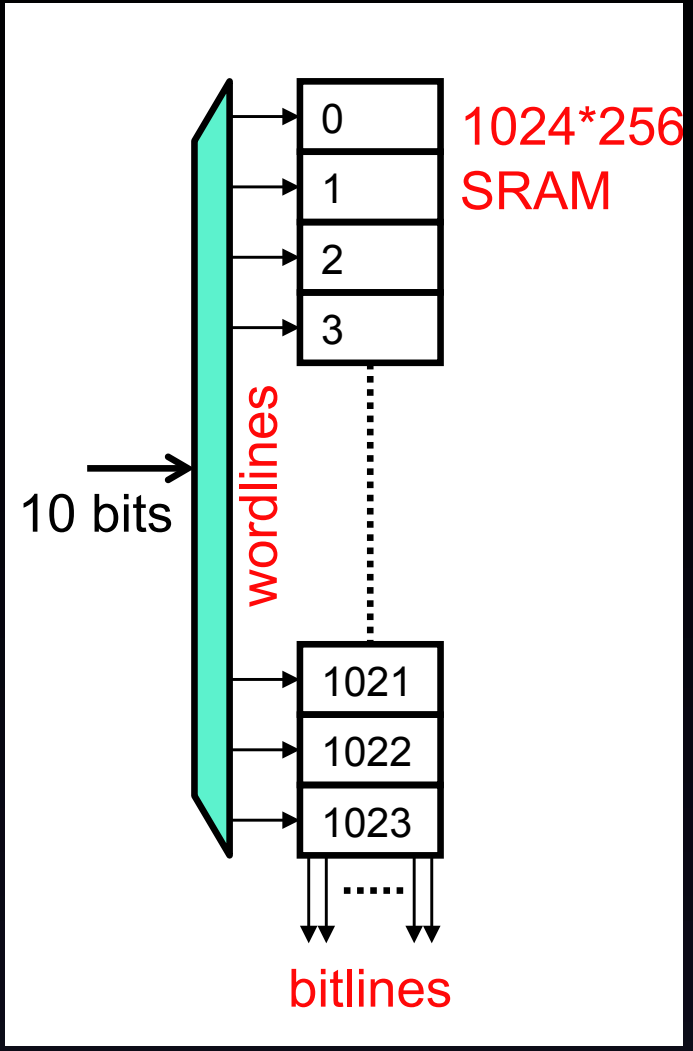
Today's Focus : Caches

- Caches : hardware managed
  - hardware automatically retrieves missing data
  - built from SRAM
- Organization
  - Array-based
  - Miss classification
- Optimization techniques
  - reducing misses
  - improving miss penalty
  - improving hit latency

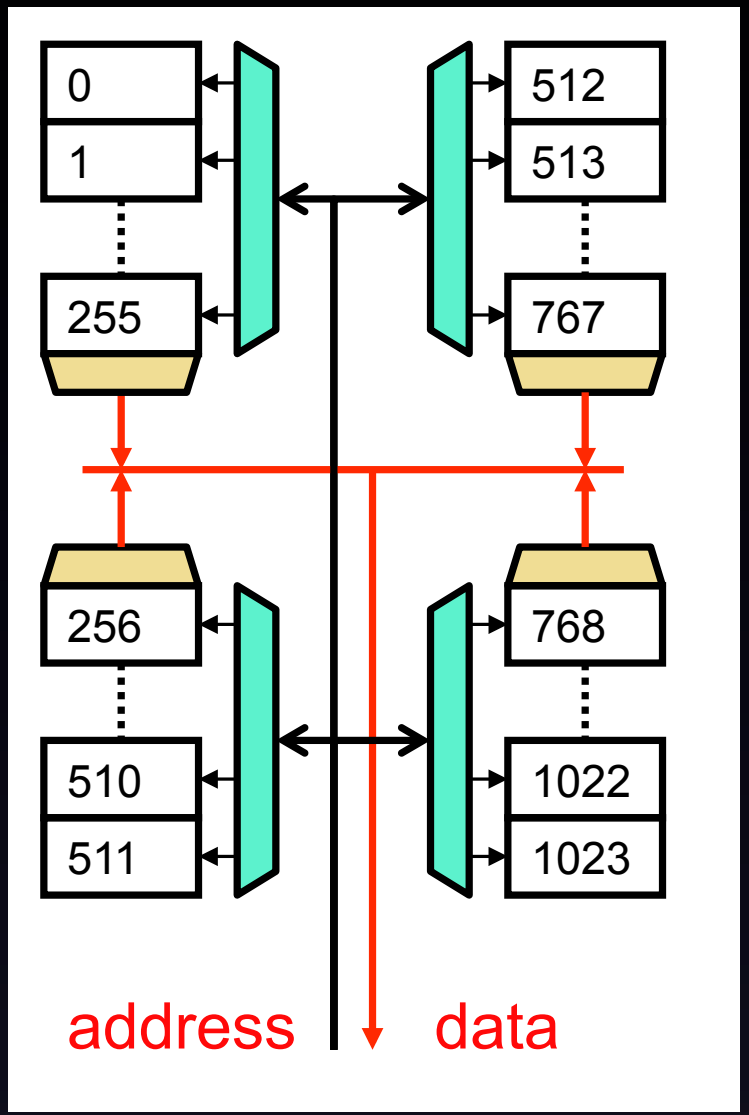


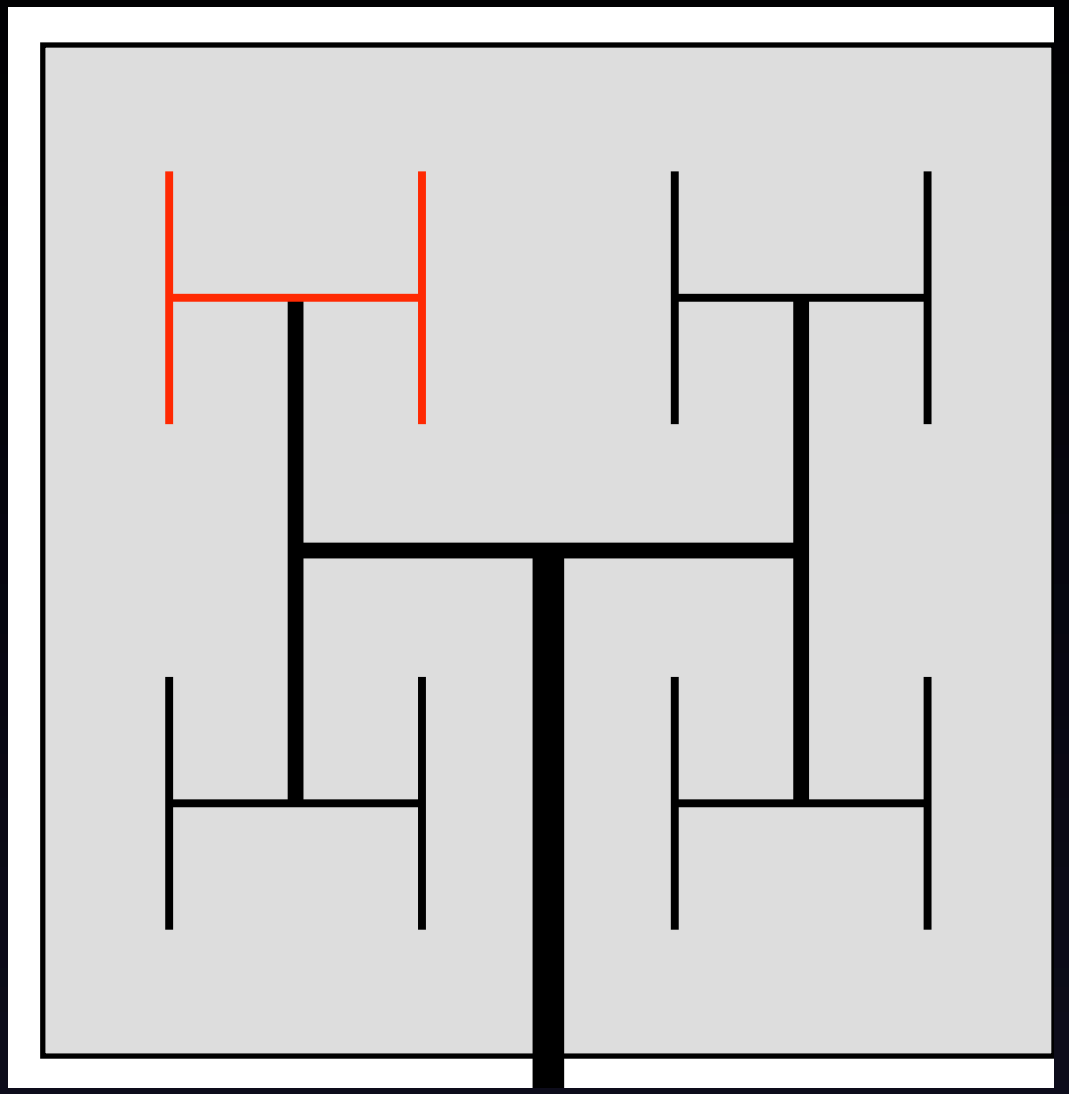
Basic Memory Structure

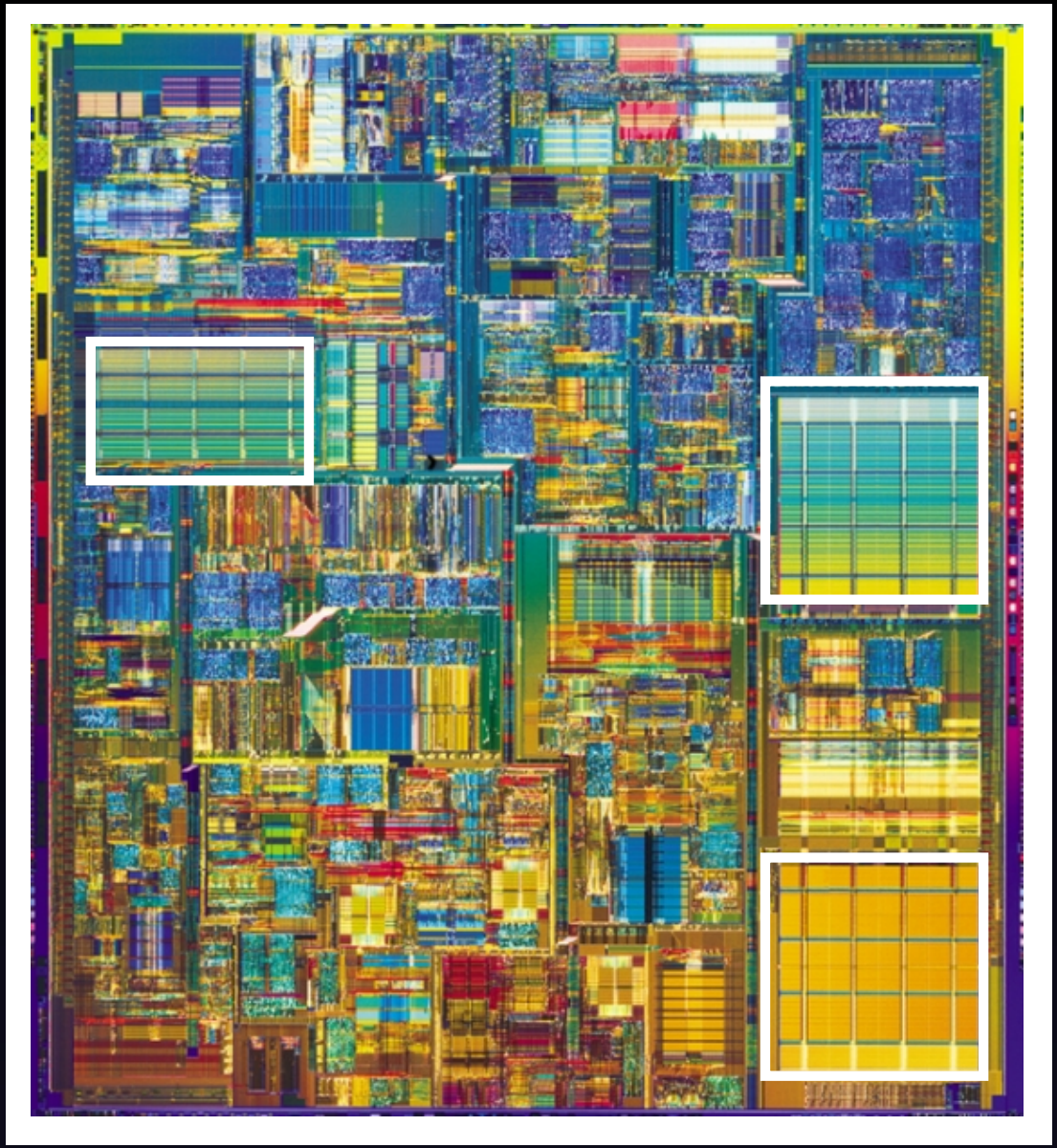
- Number of entries = 2n : n # of address bits
  - 10 bit address; 1024 entries
  - Decoder does one-hot mapping
  - address travels horizontally (word lines)
- Size of entries
  - data width access
  - data travels vertically (bit lines)



Physical Layout : H-tree







Caches

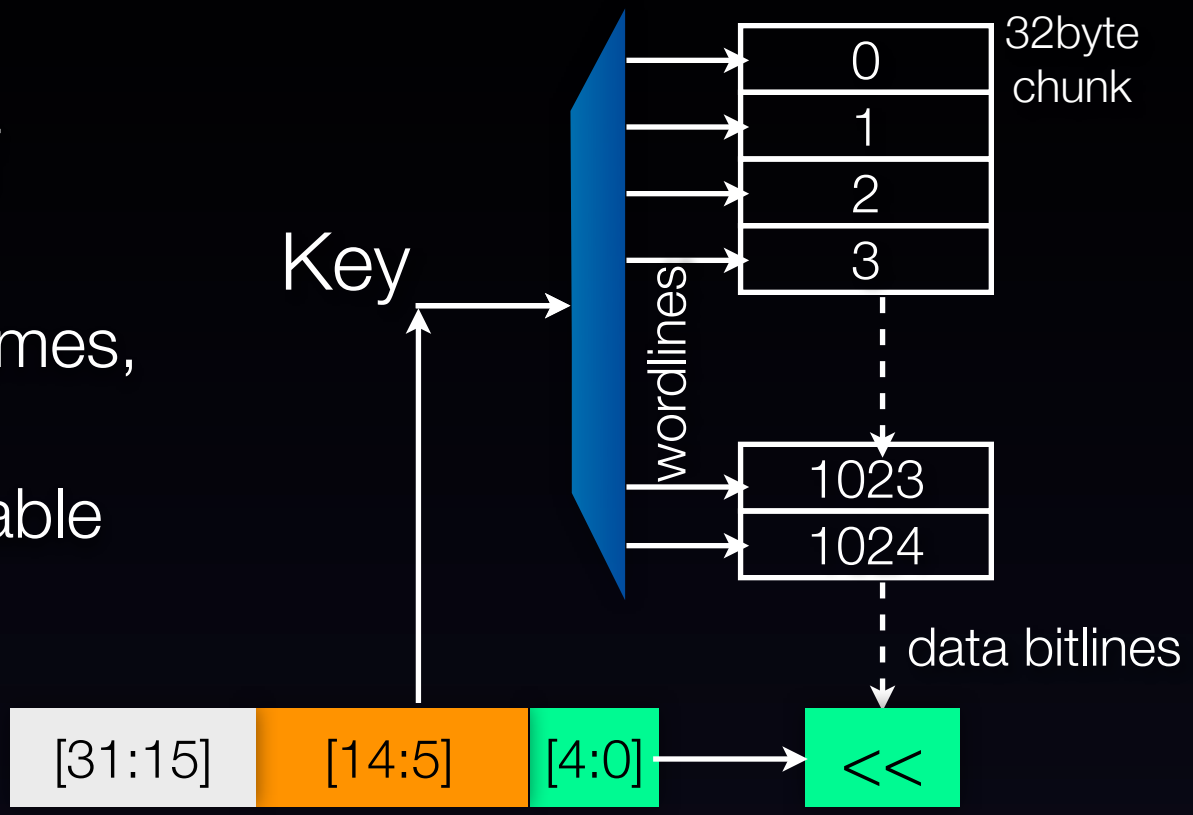
Limited, Fixed Capacity

Need to access in a fast manner

Insertion and removal should be easy

Caches Structure: Hash-Table

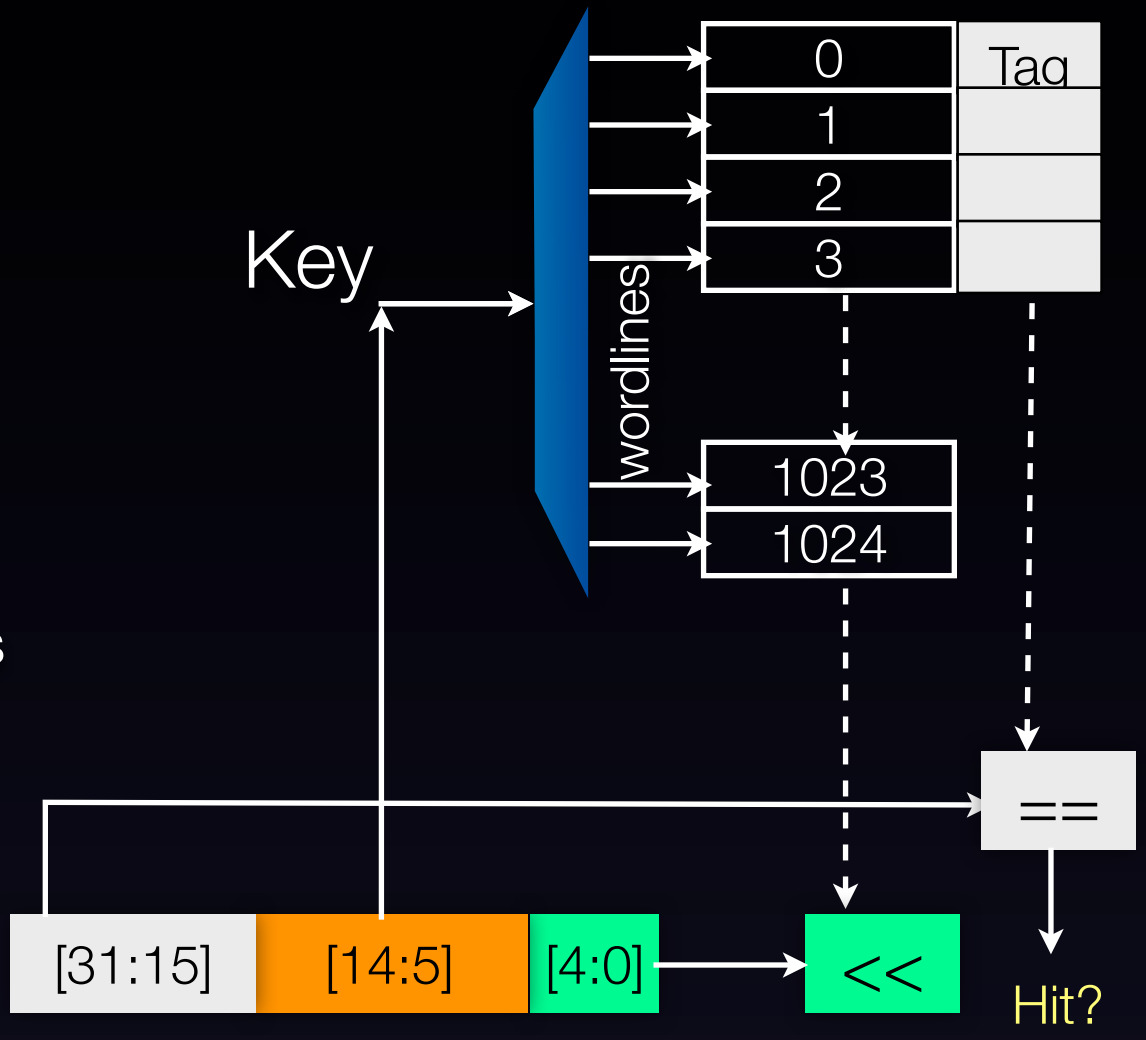
- Basic cache : Array of word chunks
  - 32KB cache (1024 frames, 32B/block)
  - Bounded-size Hash Table
- Hash-Table Key
  - 32 bit address; Max Mem. 4GB (128Million, 32byte blocks)



Why use these bits for Key ?
Hint: Hash function and spatial locality ?

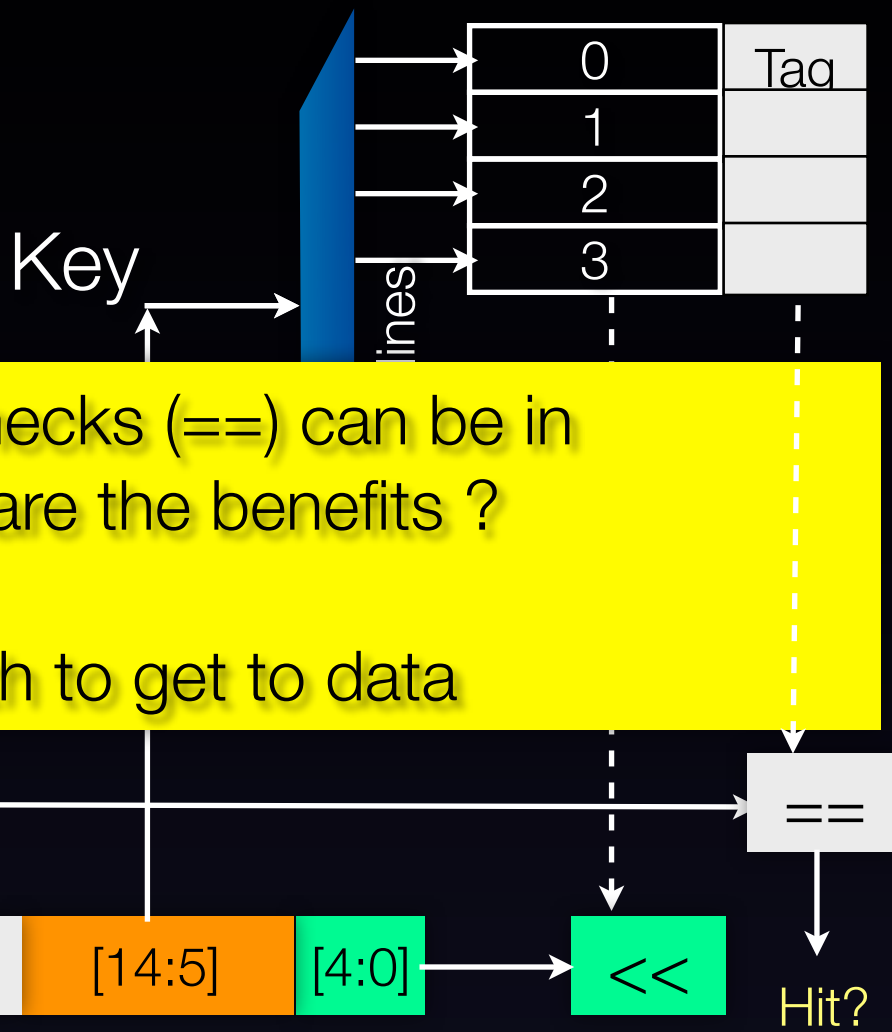
How to know you found it?

- Each Frame can hold one of the 2^{17} blocks
  - How to find out what you have cached
- Cache Tag
  - To each frame attach bits attach [31:15]
  - compare incoming address and address stored
- Overall Algorithm
  - read frame from row indicated by index
  - “Hit” if tag matches



How to know you found it?

- Each Frame can hold one of the 2^{17} blocks
  - How to find out what you have cached



Data read (<<) and Key checks (==) can be in parallel or serial? What are the benefits ?

Hint : Think critical path to get to data

- Overall Algorithm
  - read frame from row indicated by index
  - "Hit" if tag matches

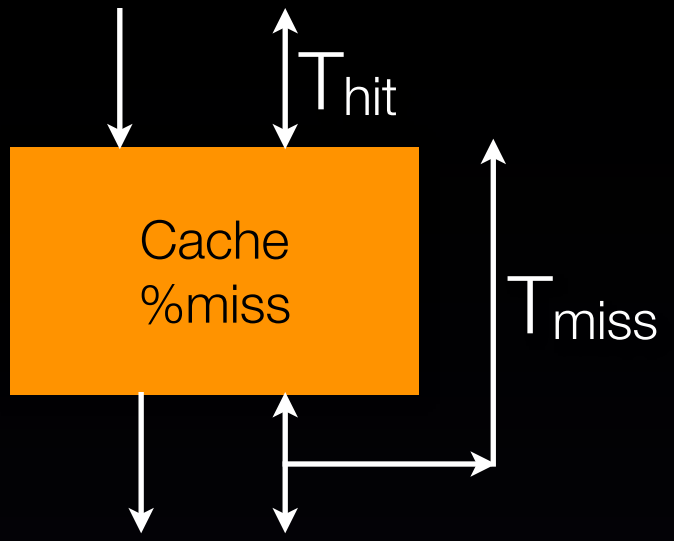
Tag overhead

- 32KB Cache = 32KB of Data storage
  - Tag storage considered to be overhead
- 32KB Cache, 1024 frames 32Bytes/frame
  - 32Byte frame (5 bit offset); 1024 frames (10bit index)
  - Max Physical Memory in system = 1 TB (40 bits)
  - Tag - 40-(5+10) = 25 bits + 1 Valid bit
    (~3.3KB storage) = 9% overhead
- If Max physical memory = 256TB (48 bits)
  - Tag overhead = 13%

Cache Misses

- What if data isn't in the cache
- Cache controller : State machine
  - remember cache miss address
  - issues message to next-level of memory
  - waits for data response
  - fills cache entry

Cache Terminology



Access : Read/Write to cache

Hit : Desired data in cache
Miss : Desired data not in cache
Fill : Data placed in cache

% miss = #misses/ #accesses
MPKI = # misses/ 1000 inst.
T_{hit} = Time to read (write) data from cache
T_{miss} = Time to read data into cache

$$T_{avg} = T_{hit} + \%miss * T_{miss}$$

Performance Calculation (Time/Ins)

- Parameters
  - Simple in-order pipeline with CPI=1
  - Instruction mix=30% loads and stores
  - D$ % miss = 10% T_{miss}= 10 cycles

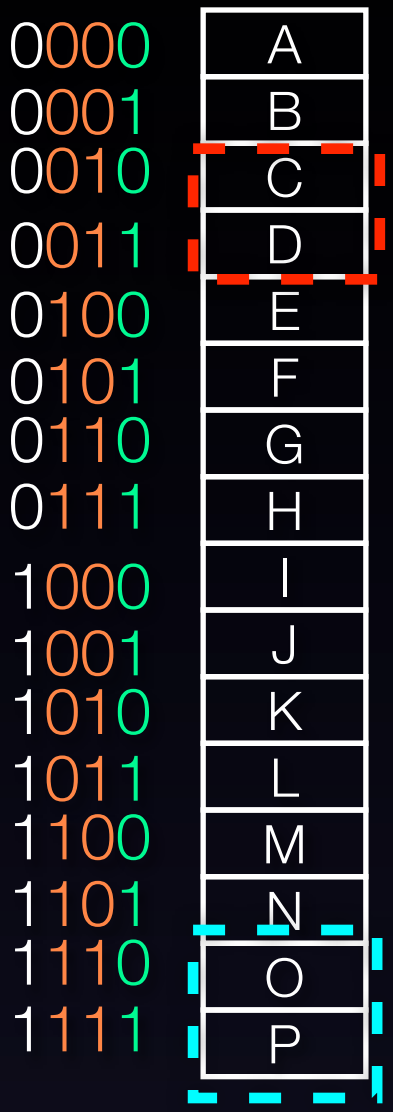
- Overall CPI
  - $CPI_{D\$}$ = %mem. access+%miss*$t_{missD\$}$
  - 0.3 cycle
  - Overall CPI = CPI + $CPI_{D\$}$ = 1.3 cycles/ins (30% higher latency if only 10% missed)

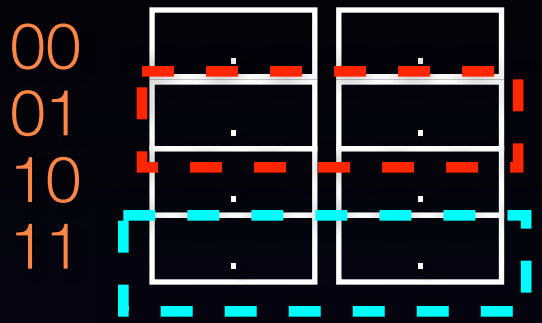
Cache Example

- 4 bit address = 16B memory
- 8B cache blocks, 2B blocks

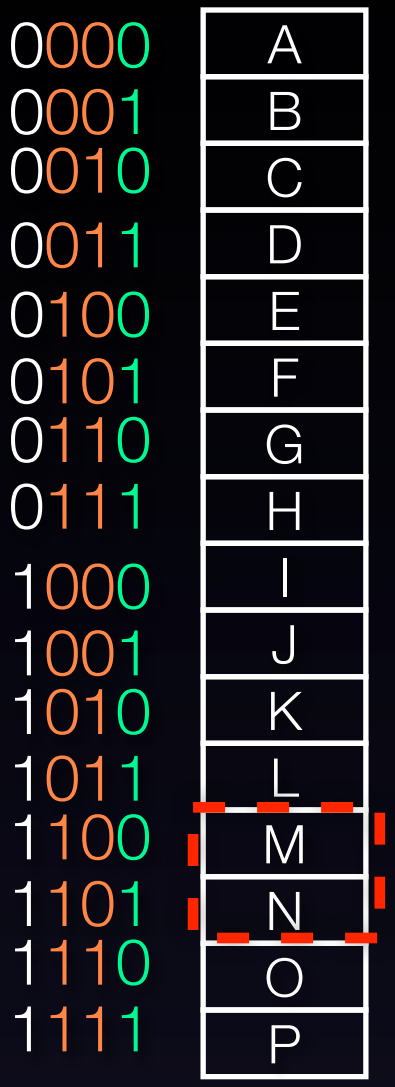
  - # of sets 4
  - Offset : least significant log (block size) = 1
  - Index : log (# sets) = 2
  - Tag rest

4 bit address, 8B cache, 2B blocks

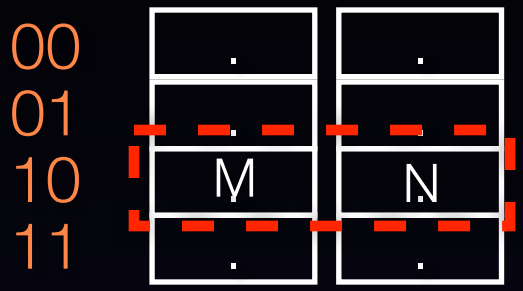




4 bit address, 8B cache, 2B blocks

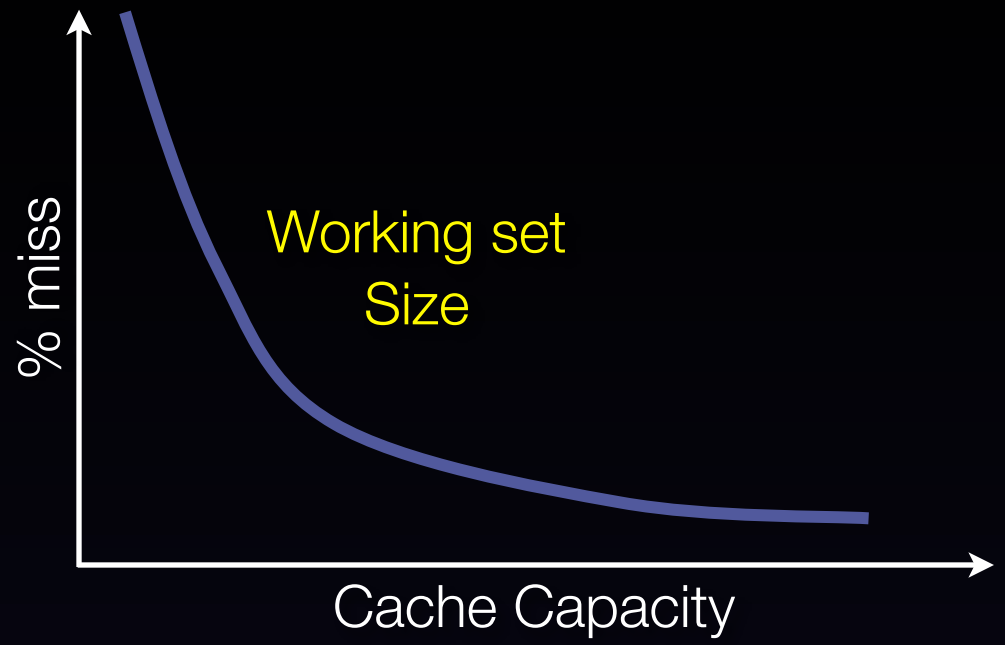


Ld 1100 (miss)



Capacity and Performance

- Reduce % miss
  - increase cache cap.
  - miss rate dec. always



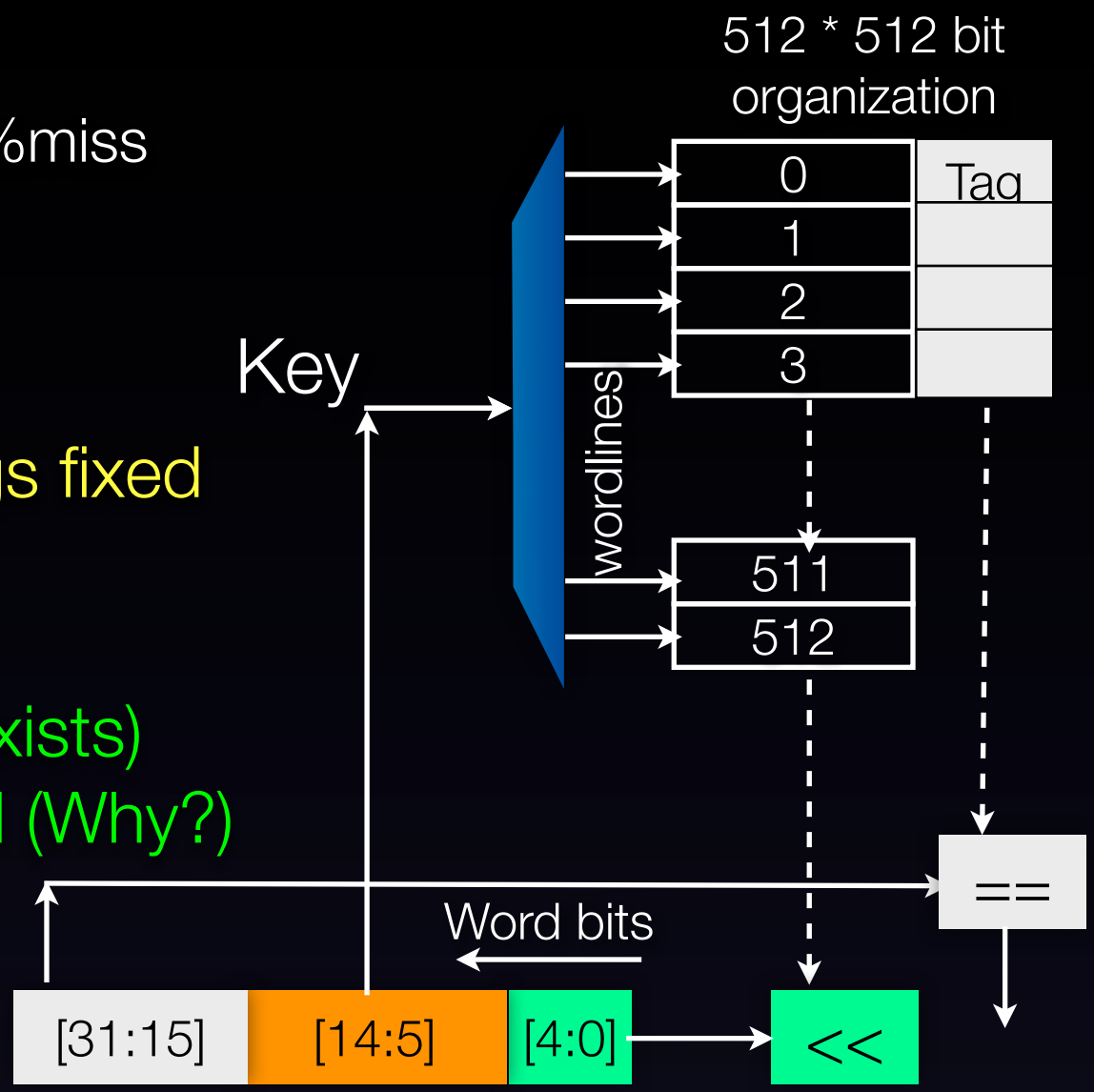
- diminishing returns

T_{hit} increases. proportional to sqrt (capacity)

Why sqrt ?

Block size

- Fixed capacity, decrease %miss
- Increase block size
  - Exploit spatial locality
  - Tag overhead remaings fixed
- Reduce miss (if locality exists)
- Tag % overhead reduced (Why?)
- Potentially wasted data transfer
- Potentially wasted storage

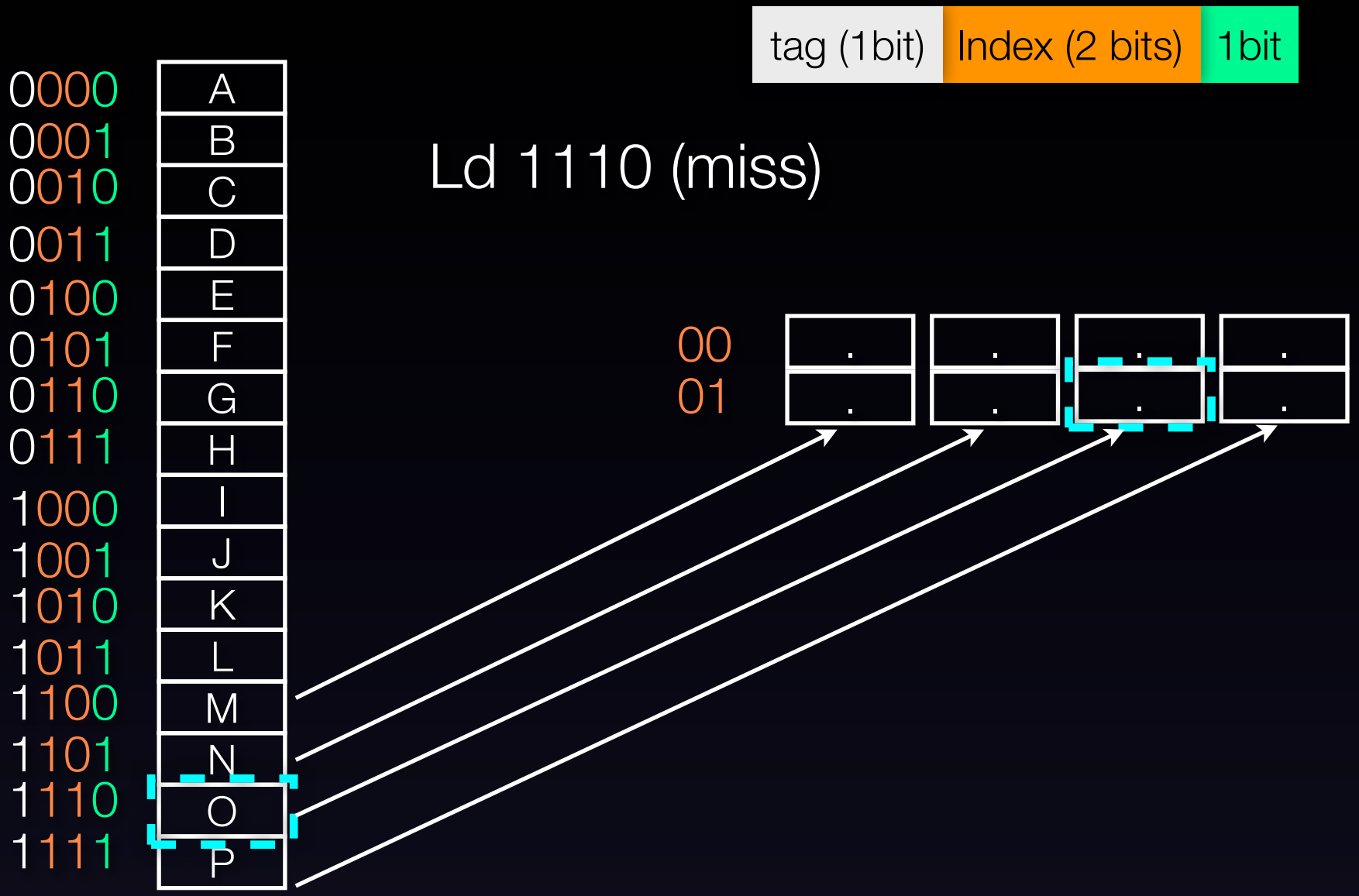


Block size and Tag overhead

- 1024 frames 32Bytes/frame (32KB)
  - 32Byte frame (5 bit offset); 1024 frames (10bit index)
  - Max Physical Memory in system = 1 TB (40 bits)
  - Tag - 40-(5+10) = 25 bits + 1 Valid bit (~3.3KB storage) = 10% overhead

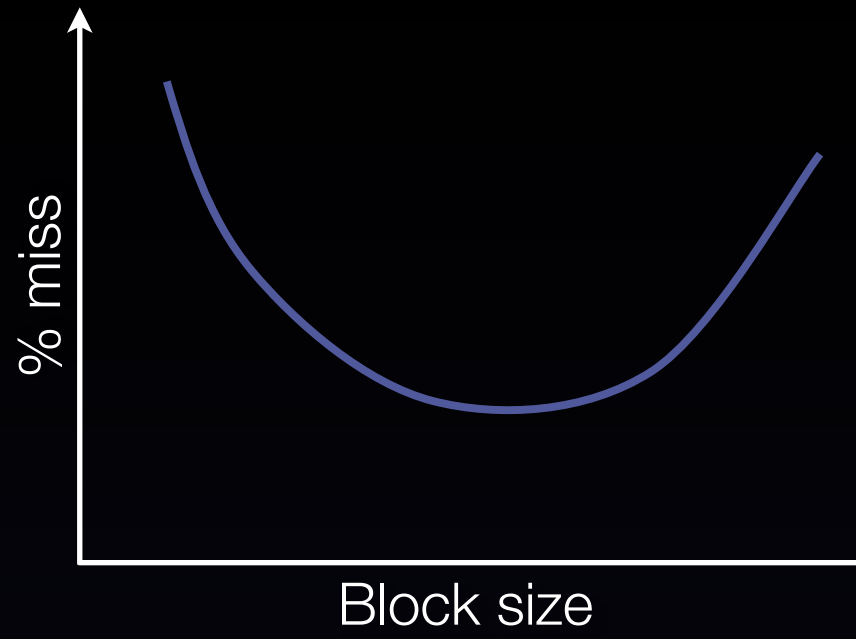
- 512 frames 64Bytes/frame (32KB)
  - 64 byte frame (6 bit offset) : 512 frames (9bit)
  - Tag = 40-(6+9)= 25 bits + 1 Valid bit
  - 26 bits/tag and 512 tags = ~1.6KB storage
  - 5% overhead

4 bit address, 8B cache, 4B blocks



Block Size and Performance

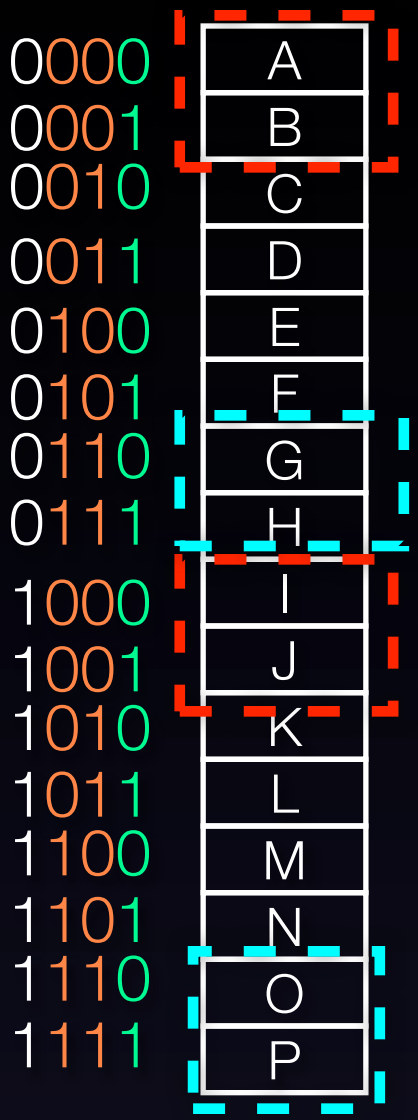
- Dual effects on miss rate
- Spatial Prefetching (good)
  - adjacent data brought in
  - misses turned into hits
- Interference (bad)
  - useful words n in different blocks
  - turns misses into hits ((limited # of unique blocks)



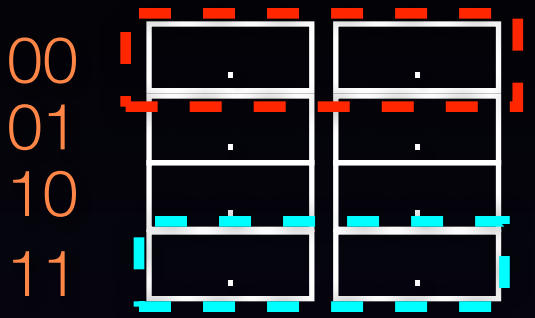
Block size and Miss penalty

- Increasing block size increases T_{miss}
  - Larger blocks take longer to transfer and refill
- However, Tmiss of invidual word not affected
  - Critical word first (req. word sent first, CPU continues)
  - remaining word refilled in the background
- T_{misses} of cluster suffers
  - more than one miss can't be handled at the same time
  - latencies affected by bandwidth (more than one miss)

Conflicts



Ld 1110 (miss)

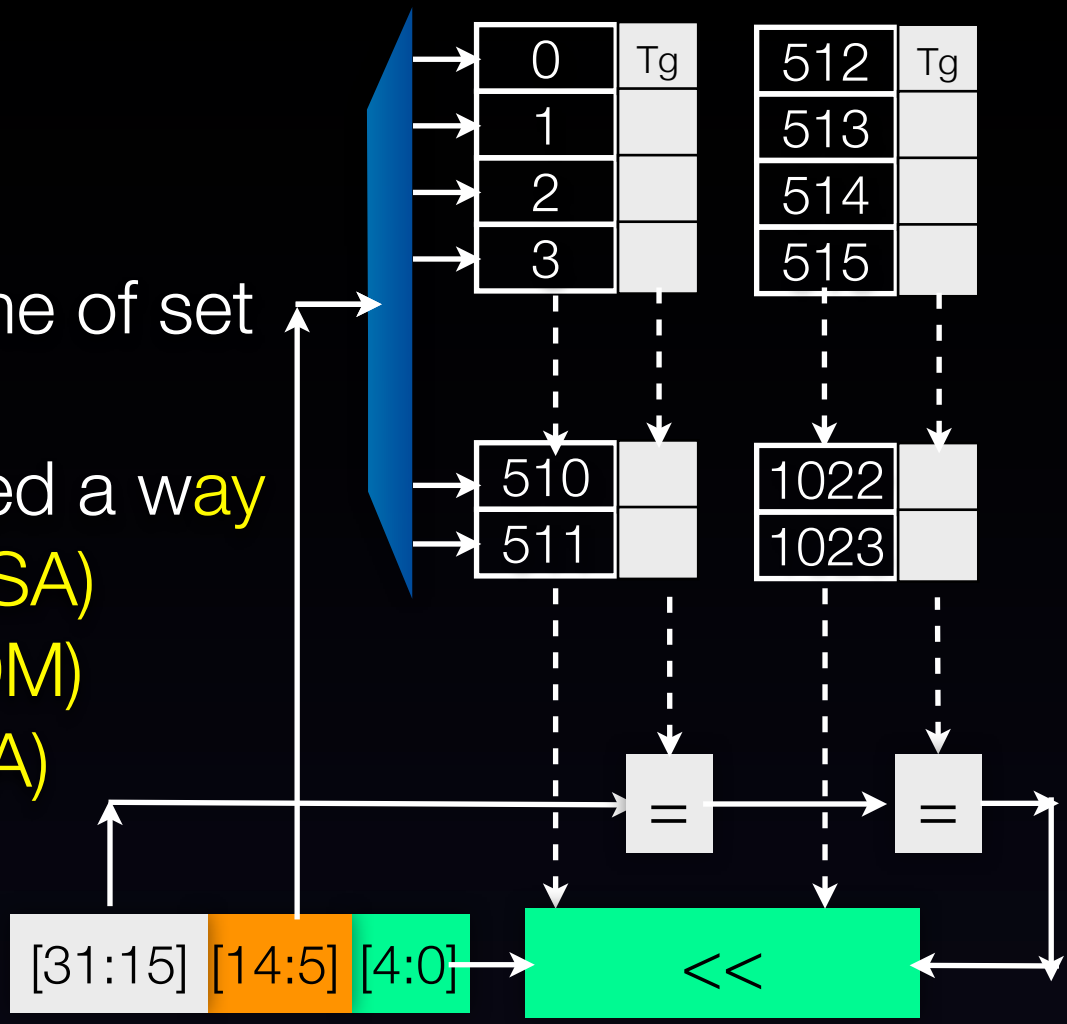


Pairs like 1110 conflict (same index)
0110

Can such pairs reside at the same time?

Set Associativity

- Set Associativity
  - Block can be in any frame of set
  - Group of frames is a set
  - Each frame in set is called a way
  - E.g., 2-way set-assoc. (SA)
  - 1-way direct-mapped (DM)
  - 1-set fully-associative (FA)

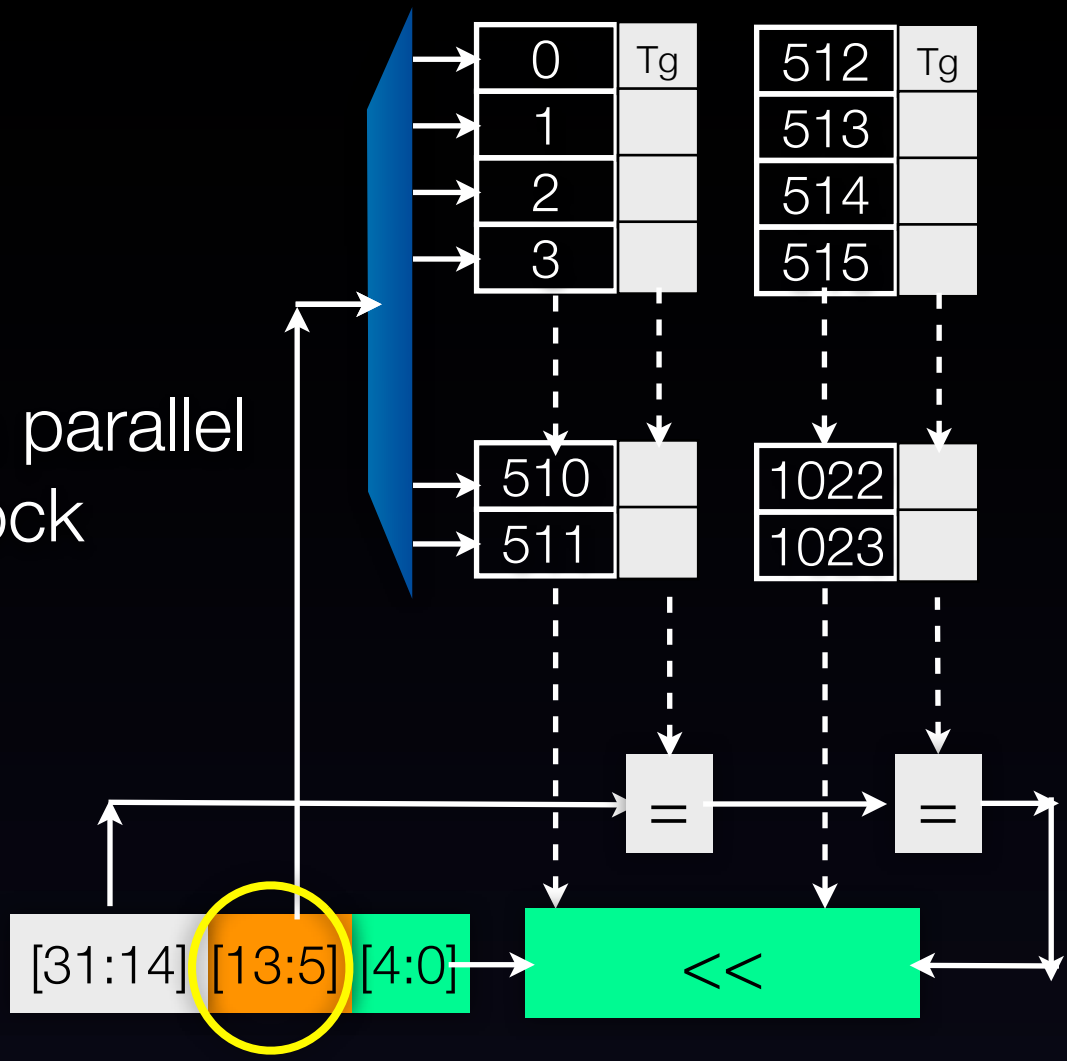


Reduces conflicts

Increases T_{hit}

Set Associativity

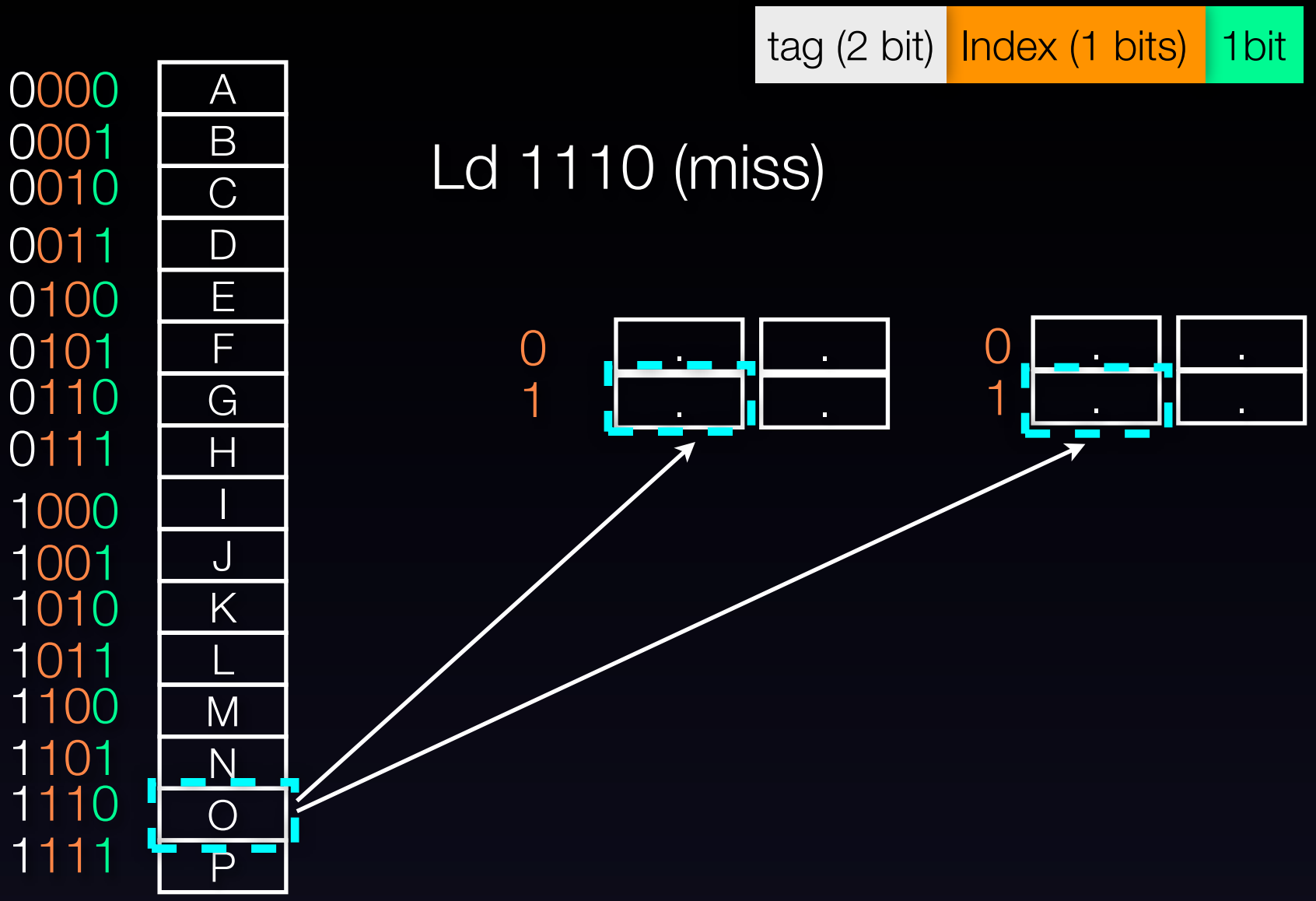
- Lookup Algorithm
  - Index bits find the set
  - Read all data frames in parallel
  - Any frame can hold block
- Tag/Index bits change
  - Only 9 bits



Replacement Policies

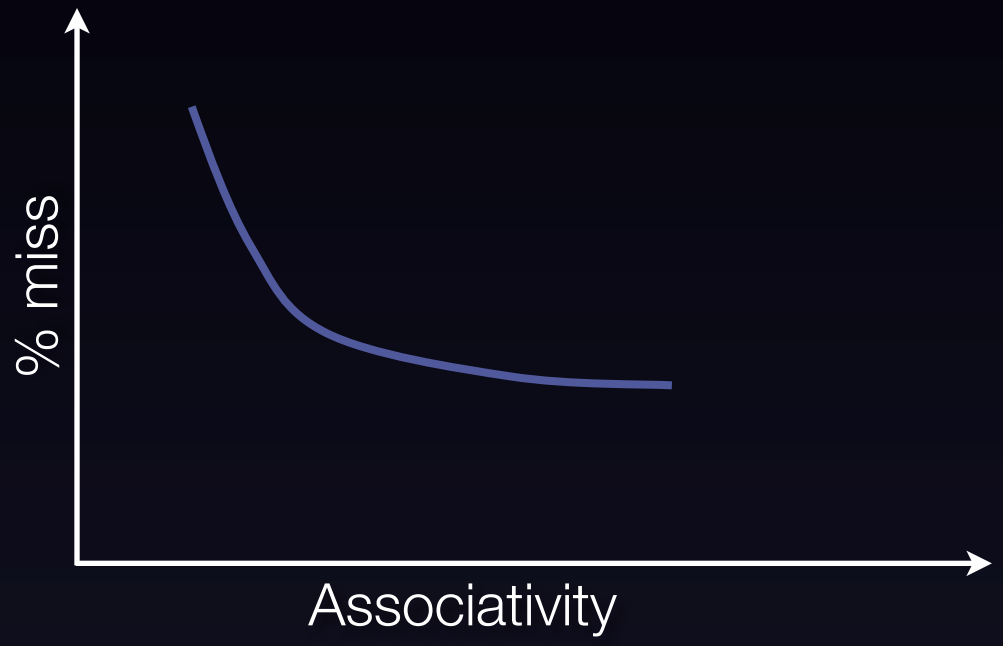
- Set-Associative caches present new challenges
  - on a cache miss, which block to replace?
- Options
  - Random, FIFO (First-in-First-Out)
  - LRU (Least recently used)
  - NMRU (Not most recently used)
  - Most optimal (Not doable)
- Replacement metadata updated on each miss

8B Cache, 2B blocks, 2Way



Associativity and Performance

- Higher Associativity
  - Lower %miss (diminishing return)
- T_{hit} increases
  - higher associativity (slower)
  - more power

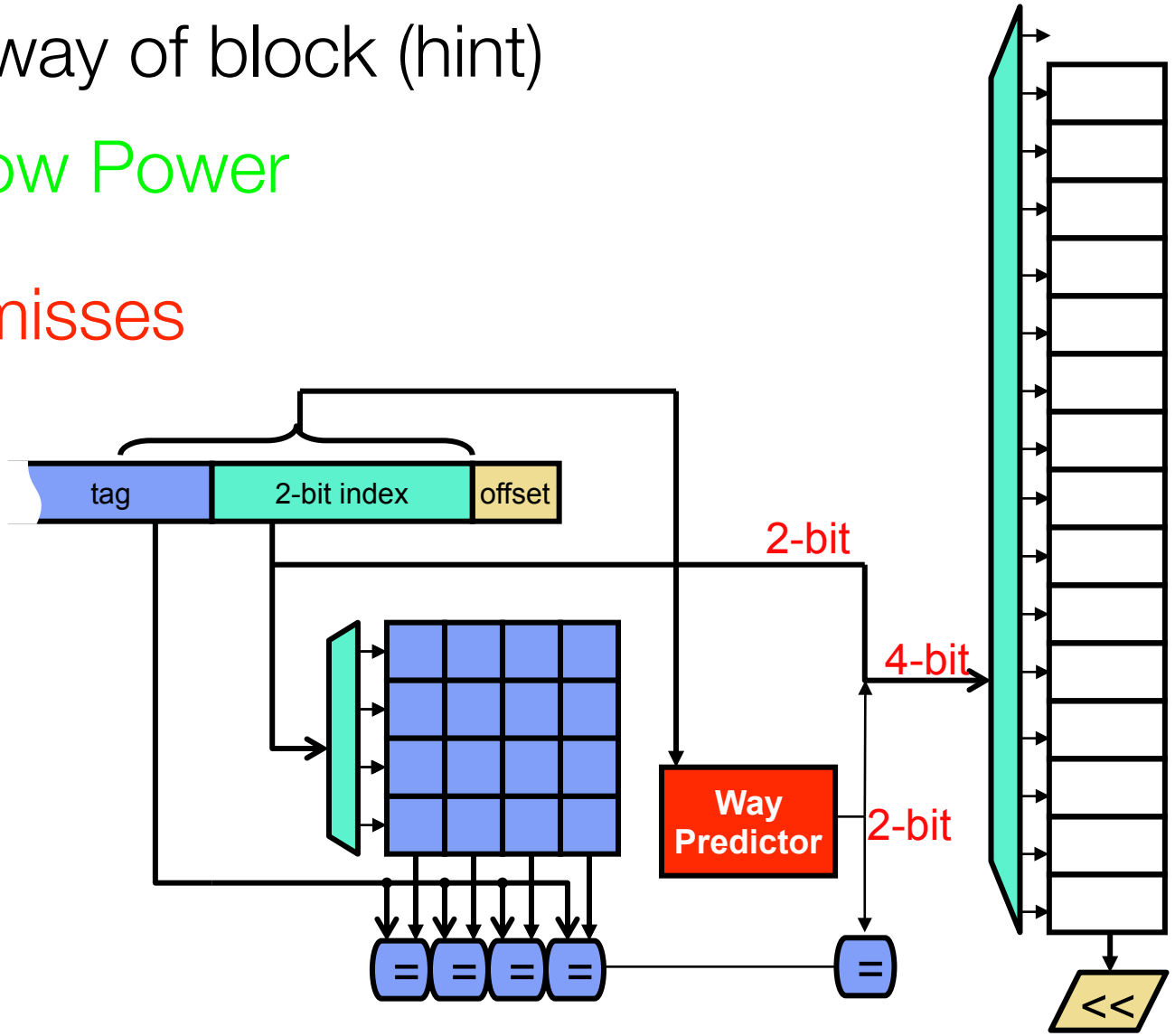


Way Prediction

Predict way of block (hint)

Fast; Low Power

More misses



Classification of Misses

- Compulsory (cold): never seen this address
  - Would miss even in infinite cache
- Capacity: miss because cache is too small
  - Would miss even in fully associative cache
  - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
- Conflict: miss caused because cache
  - associativity is too loo low. Identify? other misses
- Coherence Misses : In Multiprocessors

Miss rate Factors

Associativity

- Decreases conflict misses
- increases latency T_{hit}

Block Size

- Decreases compulsory/capacity misses (spatial locality)
- Increases conflict/capacity misses (fewer frames)
- no effect on latency T_{hit}

Capacity

- Decreases capacity misses
- increases latency of T_{hit}

Software Restructuring : Data

- Capacity misses: poor locality
  - code restructuring

- Loop interchange: spatial locality
  - Row major matrix X[i][j] followed by X[i][j +1]Poor code X[i][j] followed by X[i+1][j]

```
for (j = 0; j<NCOLS; j++)
   for (i = 0; i<NROWS; i++)
      sum += X[i][j];
```

- Better Code

```
for (j = 0; j<NROWS; j++)
   for (i = 0; i<NCOLS; i++)
      sum += X[i][j];
```

Software Restructuring: Data

- Loop blocking: temporal locality
- Poor code

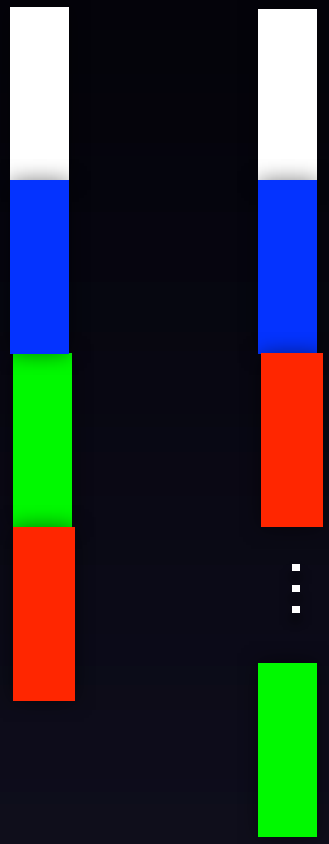
```
for (k = 0; k<NITERATIONS; k++)
   for (i = 0; i<NELMS; i++)
      sum += X[i];
```

- Better code
  - cut array into CACHE_SIZE chunks
  - run all phases on one chunck

```
for (i = 0; i<NELEMS; i+=CACHE_SIZE)
   for (ii = 0; ii<i+CACHE_SIZE-1;ii++)
      sum += X[ii];
```

Software restructuring code

- Compiler can layout code for ins. locality
  - If (a) {code 1;} else {code 2;} code 3;
  - code 2 never happens



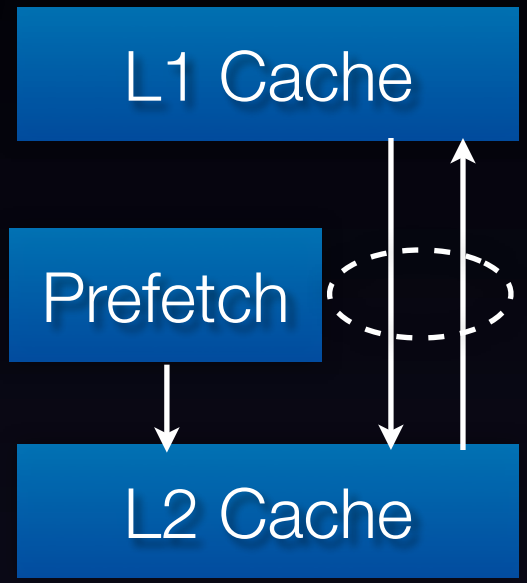
Fewer branches, too
Intra-procedure, inter-procedure

Java virtual machine does this

Prefetching : Speculation

- Proactively fetch data chunks into cache
  - need to predict/anticipate upcoming miss addresses
  - can be done in hardware or software

- Next block prefetcher (Intel L1 and L2)
  - Miss on address X => fetch X+1
  - Works for ins.: sequential execution
  - works for data arrays



- Design choice
  - Timeliness: Initiate prefetches
  - Coverage: Prefetch as many misses as possible
  - Pollution: Unnecessary data

Software Prefetching

- "Prefetch" (read) and "PrefetchW" (write)
  - read data into cache not register (Why?)
  - No guarantees
  - Inserted by programmer or compiler

```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j+=BLOCK_SIZE) {
      __builtin_prefetch(&X[i][j]
  +BLOCK_SIZE);
      for (jj=j; jj<j+BLOCK_SIZE-1; jj++)
         sum += x[i][jj];
   }
```

- Multiple prefetches using multiple blocks (in parallel)
  - more "memory-level" parallelism (How does it help?)

Hardware Prefetches: Intel

- What to prefetch? strides and other patterns
- Stride-based sequential prefetching
  - works for many common patterns; inst. and arrays
  - exploits spatial locality without inc. block size
- Address prediction
  - more complicated data structures; trees, list etc
  - record, trigger and replay

Writing to the cache

- Multiple design choices
  - Cache access
  - Write-through vs. Write-back
  - Write-allocate vs. Write not-allocate
  - How to hide write latency?

Tag/Data Access

- Read: read tag and data in parallel
  - tag is wrong; wait for for data
- Writes; read tag, write data in parallel? Why?
  - Tag mis-match -> data is mutated
- Writes are completed in two-stages
  - Step 1: match tag
  - Step 2: write to matching block

Write Propagation

- When to propagate new value to memory?
- Option 1: Write-through immediately
  - on hit, write data to cache
  - on miss, send write value to next level

- Option 2: Write-back, when block is replaced
  - track which blocks are written
  - when dirty (written) block; write to next level

- Writeback-buffer
  - #1: send “refill” request to next level
  - #2: when waiting, write block to buffer
  - #3: write value to cache and to next level

Wr-Through vs Wr-Back

- Write-through

  No writeback HW Simple design

  Extra bandwidth (if same variable written repeatedly)

  Too many small writes (1-8bytes)

  Sun Niagara, IBM Power for L1 cache

- Write-back

  Amortize write overhead Less bandwidth

  Used by Intel and AMD for all cache levels

Write-miss Handling

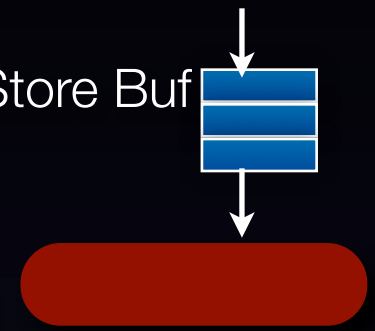
- What to do on write miss?
- Write Allocate : refill from next level
  - improved hits (read to written data)
  - Extra bandwidth (get data you may not need)
  - Writeback cache
- Write No-Allocate
  - Uses less bandwidth
  - Writethrough cache

extra write misses

Store buffer

- Read miss? Load has to wait for data
- Write miss? No need to wait

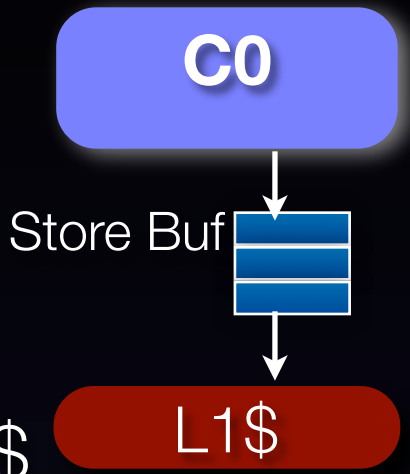
- Store buffer: core writes data to it
  - frees up processor to do other work
  - elimates stall on write misses
  - loads's data can be in either Store-Buf or L1$



- Store buffer vs Writeback buffer
  - Writeback: behind L1$ for hiding writeback
  - Store buffer: in front of caches for freeing up core

Store buffer

- Read miss? Load has to wait for data
- Write miss? No need to wait



- Store buffer: core writes data to it
  - frees up processor to do other work
  - elimates stall on write misses
  - loads's data can be in either Store-Buf or L1$

- Store buffer vs Writeback buffer
  - Writeback: behind L1$ for hiding writeback
  - Store buffer: in front of caches for freeing up core

Designing a cache hierarchy

- Tradeoff T_{hit} vs % miss tradeoff
- Upper components (I$,D$) emphasize low T_{hit}
  - Frequent access => T_{hit} important.
  - t_{miss} not high => % miss not important
  - low capacity/associativity (to reduce T_{hit})
  - small-medium block size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to %miss
  - Infrequent access => % miss important
  - T_{miss} => % miss important
  - High capacity/associativity/block size (to reduce %miss)

Memory Hierarchy Parameters

Param.	I$/D$	L2	L3	Main Mem.
T_{hit}	2ns	10ns	30ns	100ns
T_{miss}	10ns	30ns	100ns	10ms (10^6 ns)
Capacity	8KB-64KB	256KB-8MB	2-16MB	1-4GB
Block Size	16B-64B	32B-128B	32B-256B	
Associativity	1-4	4-16	4-16	

Inclusive vs Exclusive

- Inclusion (Intel)
  - Bring block from mem. into L2 and than L1
  - If block in L2, then in L1 as well
  - If block evicted from L2, then L1 as well
- Exclusion (AMD)
  - Bring block from L2 to L2 but not (L2 or lower)
  - Good if L2 cache not that big
- Non-inclusion (AMD)
  - No guarantees. First time bring only into L1
  - evict and reload, keep in both L1 and L2.

Miss rate / access vs instruction

- For Level 1 caches use instruction mix
  - If memory ops. are 1/3rd of ins.
  - 2% of inst. miss (1 in 50) is 6% of access
- For Level 2 caches
  - Misses per inst. still straightforward
  - Misses per-L2 reference more indirect
    - L1 misses = # L2 references.