

1010

Caching

ENGR 3410 - Computer Architecture

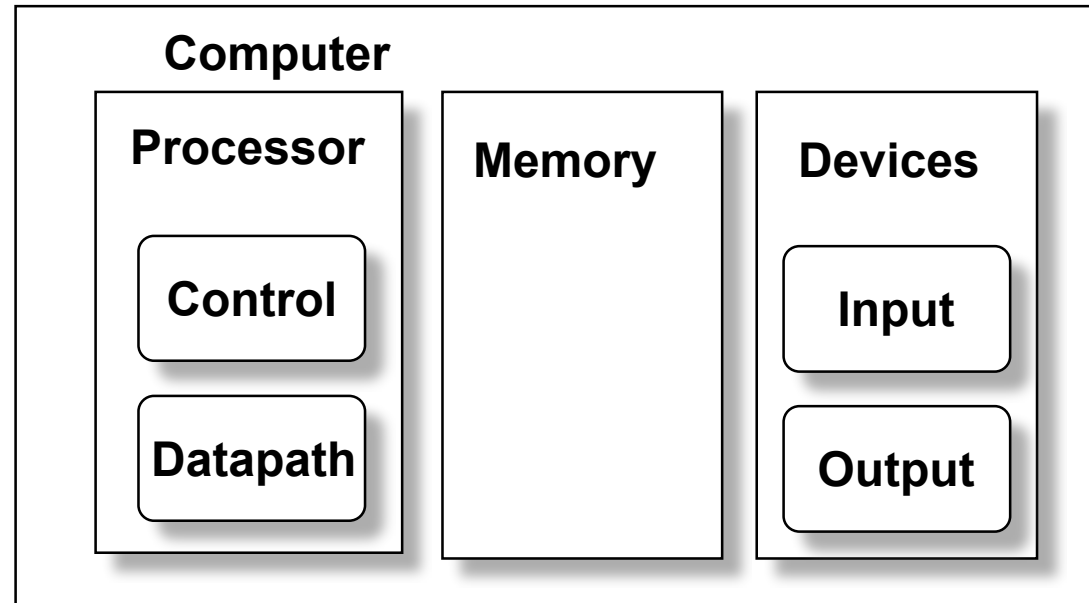
Mark L. Chang

Fall 2006

Memory Hierarchy: Caches, Virtual Memory

Big memories are slow

Fast memories are small



Need to get fast, big memories

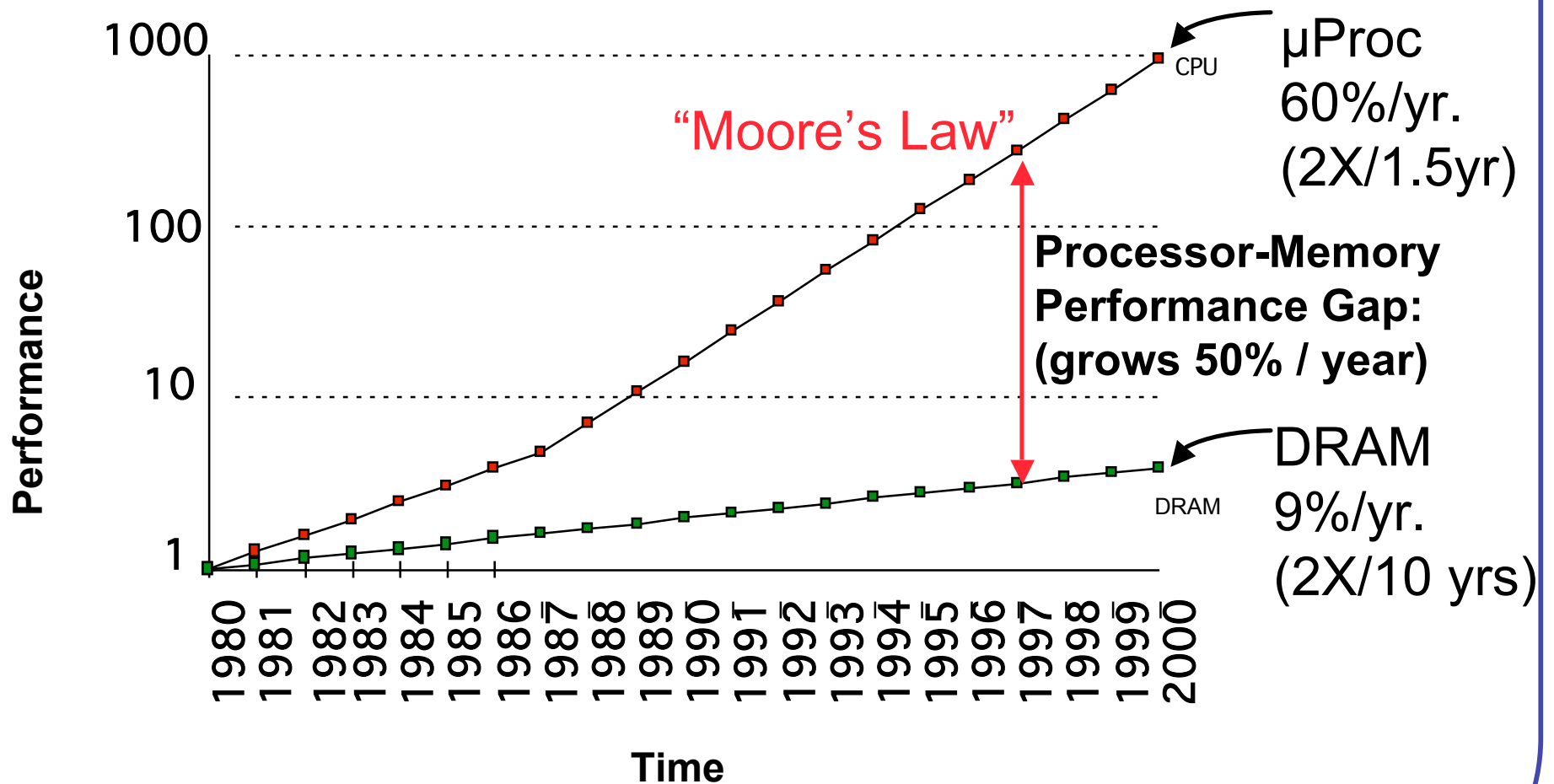
Random Access Memory

- Dynamic Random Access Memory (DRAM)
 - High density, low power, cheap, but slow
 - Dynamic since data must be “refreshed” regularly
 - Random Access since arbitrary memory locations can be read
- Static Random Access Memory
 - Low density, high power, expensive
 - Static since data held as long as power is on
 - Fast access time, often 2 to 10 times faster than DRAM

Technology	Access Time	\$/MB in 1997
SRAM	5-25ns	\$100-\$200
DRAM	60-120ns	\$5-\$10
Disk	$(10-20) \times 10^6$ ns	\$0.10-\$0.20

Technology Trends

- Processor-DRAM Memory Gap (latency)



The Problem

- The Von Neumann Bottleneck
 - Logic gets faster
 - Memory capacity gets larger
 - Memory speed is not keeping up with logic
- Cost vs. Performance
 - Fast memory is expensive
 - Slow memory can significantly affect performance
- Design Philosophy
 - Use a hybrid approach that uses aspects of both
 - Keep frequently used things in a small amount of fast/expensive memory
- “Cache”
 - Place everything else in slower/inexpensive memory (even disk)
 - Make the common case fast

Locality

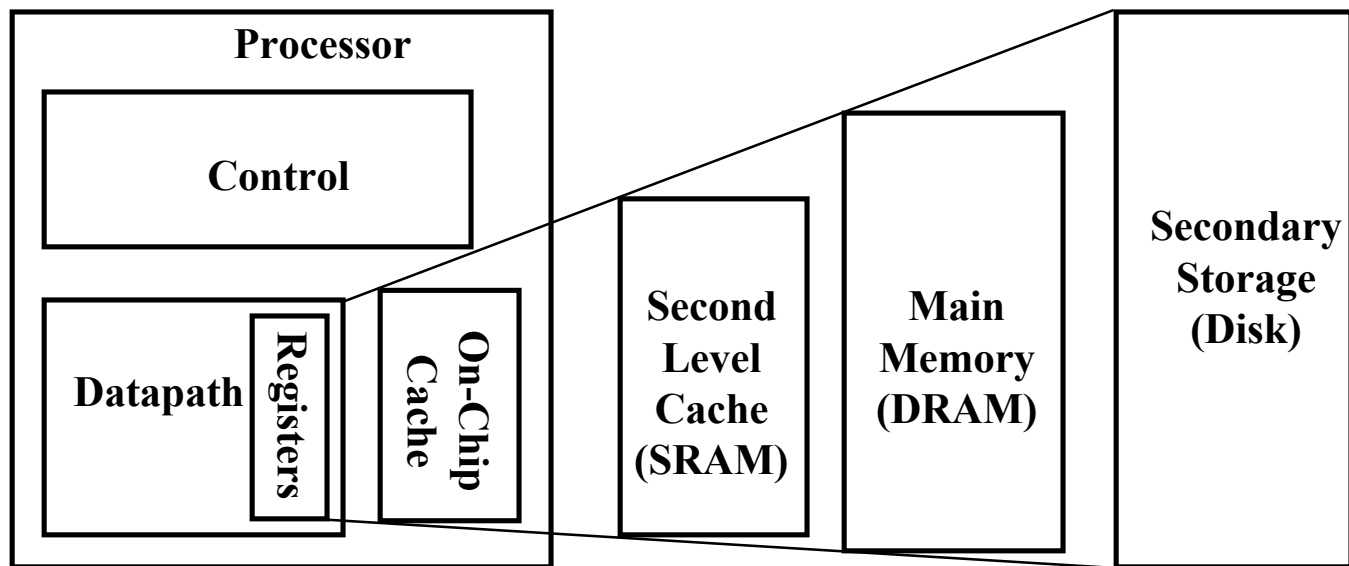
- Programs access a relatively small portion of the address space at a time

```
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= 'a' && *index <= 'z')
        *index = *index + ('A' - 'a');
    index++;
}
```

- Types of Locality
 - Temporal Locality - If an item has been accessed recently, it will tend to be accessed again soon
 - Spatial Locality - If an item has been accessed recently, nearby items will tend to be accessed soon
- Locality guides caching

The Solution

- By taking advantage of the principle of locality:
 - Provide as much memory as is available in the cheapest technology.
 - Provide access at the speed offered by the fastest technology.



Name	Register	Cache	Main Memory	Disk Memory
Speed	<1ns	<10ns	60ns	10 <i>ms</i>
Size	100 Bs	KBs	MBs	GBs

Cache Terminology

- **Block:** Minimum unit of information transfer between levels of the hierarchy
 - Block addressing varies by technology at each level
 - Blocks are moved one level at a time
- **Hit:** Data appears in a block in that level
 - **Hit rate** - percent of accesses hitting in that level
 - **Hit time** - Time to access this level
 - Hit time = Access time + Time to determine hit/miss
- **Miss:** Data does not appear in that level and must be fetched from lower level
 - **Miss rate** - percent of misses at that level = $(1 - \text{hit rate})$
 - **Miss penalty** - Overhead in getting data from a lower level
 - Miss penalty = Lower level access time + Replacement time + Time to deliver to processor
 - Miss penalty is usually MUCH larger than the hit time

Cache Access Time Example

Level	Hit Time (access)	Hit Rate	Access Time
L1	1 cycle	95%	$1 + 0.05 \times 65 = 4.25$
L2	10 cycles	90%	$10 + 0.10 \times 550 = 65$
Main Memory	50 cycles	99%	$50 + 0.01 \times 50000 = 550$
Disk	50,000 cycles	100%	50,000

- Note: Numbers are **local** hit rates - the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)

Cache Access Time

- Average access time
 - Access time = (hit time) + (miss penalty)x(miss rate)
 - Want high hit rate & low hit time, since miss penalty is large
- Average Memory Access Time (AMAT)
 - Apply average access time to entire hierarchy.

Handling A Cache Miss

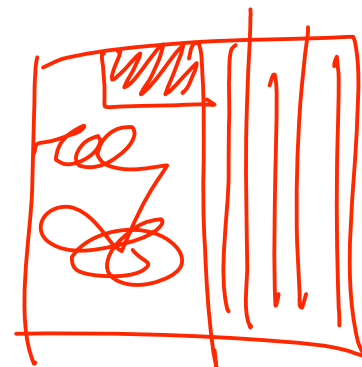
- Processor expects a cache hit (1 cycle), so no effect on hit.
- Instruction Miss
 1. Send the original PC to the memory
 2. Instruct memory to perform a read and wait (no write enables)
 3. Write the result to the appropriate cache line
 4. Restart the instruction
- Data Miss
 1. Stall the pipeline (freeze following instructions)
 2. Instruct memory to perform a read and wait
 3. Return the result from memory and allow the pipeline to continue

Exploiting Locality

- Spatial locality
 - Move blocks consisting of multiple contiguous words to upper level
- Temporal locality
 - Keep more recently accessed items closer to the processor
 - When we must evict items to make room for new ones, attempt to keep more recently accessed items

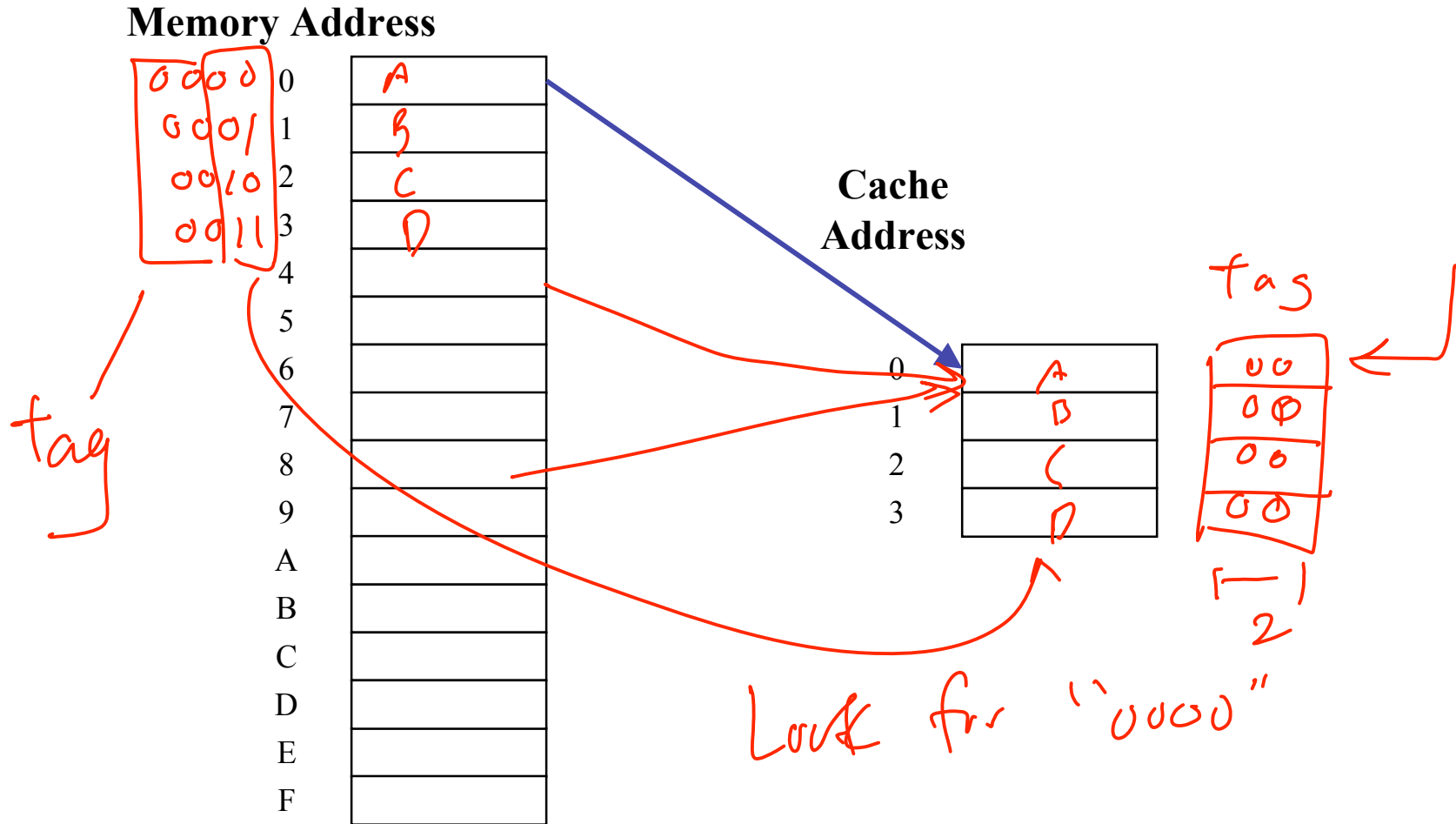
Cache Arrangement

- How should the data in the cache be organized?
Caches are smaller than the full memory, so multiple addresses must map to the same cache “line”
- **Direct Mapped** - Memory addresses map to particular location in that cache
- **Fully Associative** - Data can be placed anywhere in the cache
- **N-way Set Associative** - Data can be placed in a limited number of places in the cache depending upon the memory address



Direct Mapped Cache

- 4 byte direct mapped cache with 1 byte blocks
- Optimize for spatial locality (close blocks likely to be accessed soon)



Finding A Block

- Each location in the cache can contain a number of different memory locations

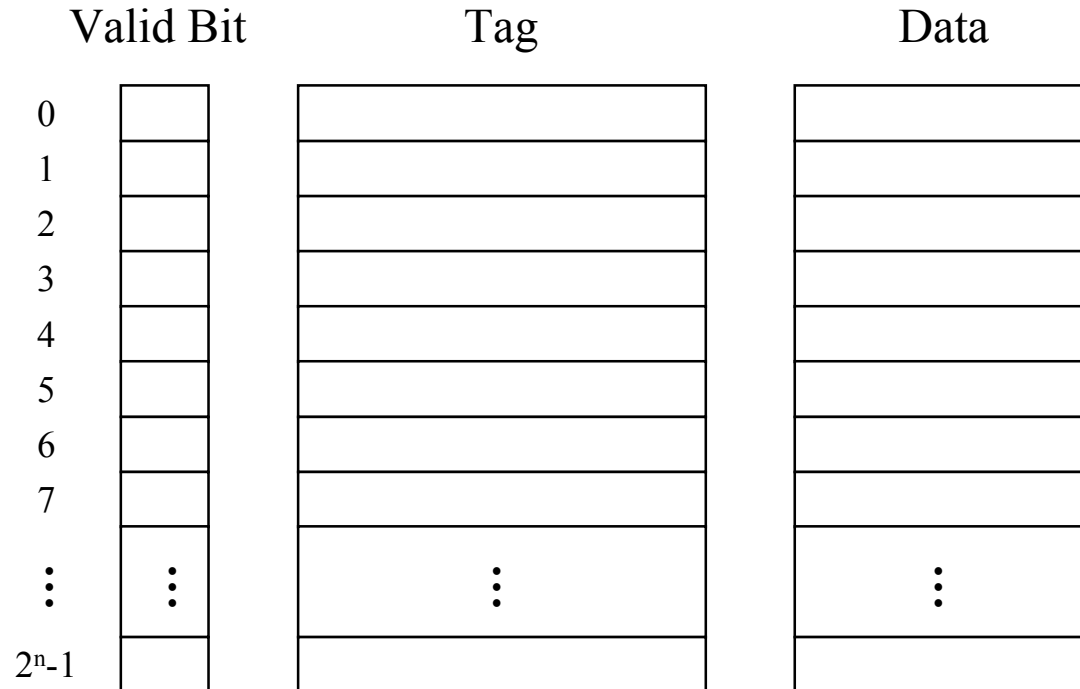
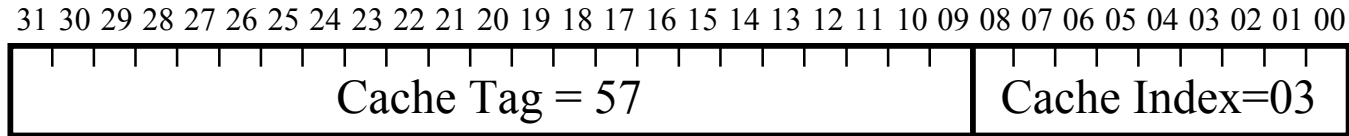
Cache 0 could hold 0, 4, 8, 12, ...

- We add a **tag** to each cache entry to identify which address it currently contains

What must we store?

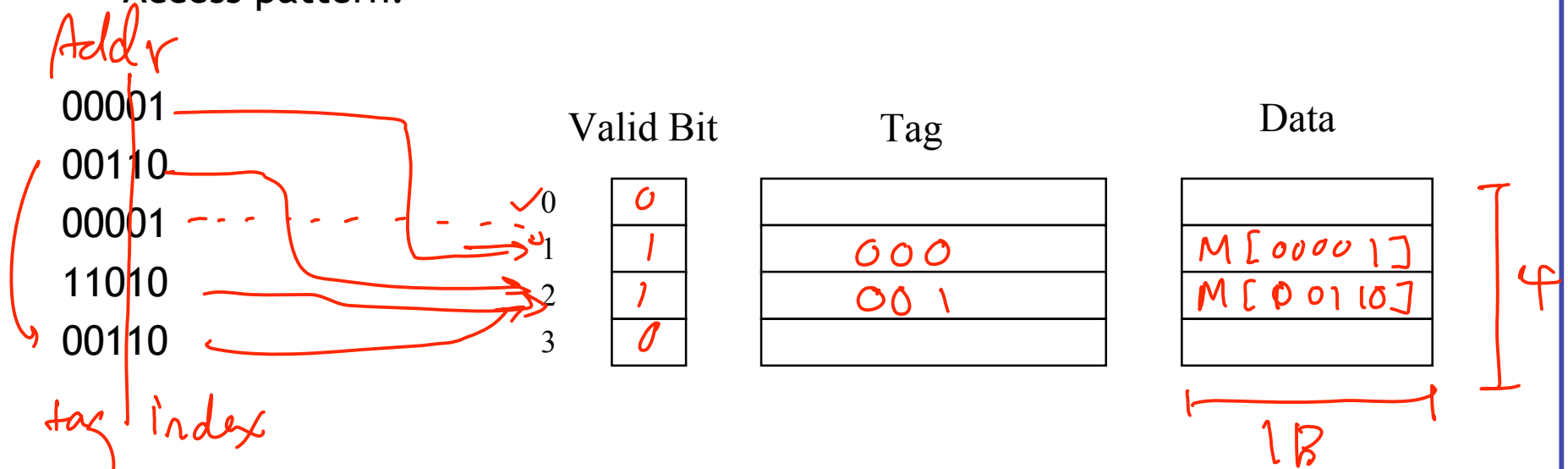
Cache Tag & Index

- Assume 2^n byte direct mapped cache with 1 byte blocks



Cache Access Example

- Assume 4 byte cache
- Access pattern:



cold
cold
hit
cold
Capacity

Mem

Cache Access Example (cont.)

- Assume 4 byte cache
- Access pattern:

00001

00110

00001

11010

00110

	Valid Bit	Tag	Data
0	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Cache Access Example (cont. 2)

- Assume 4 byte cache
- Access pattern:

00001

00110

00001

11010

00110

	Valid Bit	Tag	Data
0	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Cache Size Example

- How many total bits are required for a direct-mapped cache with 64 KB of data and 1-byte blocks, assuming a 32-bit address?

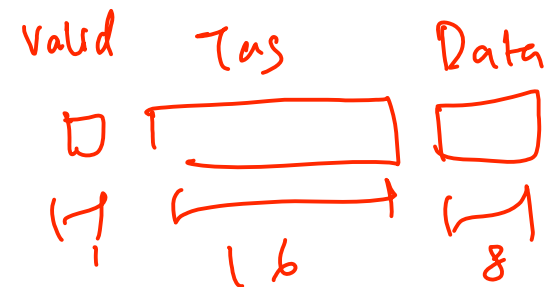
Index bits: $64\text{KB} = 2^{16}$ separate "rows" / "blocks" = 16 b index

Bits/block: 25 bits/block

Data: 8 bits

Valid: 1 bit

Tag: 16 bits

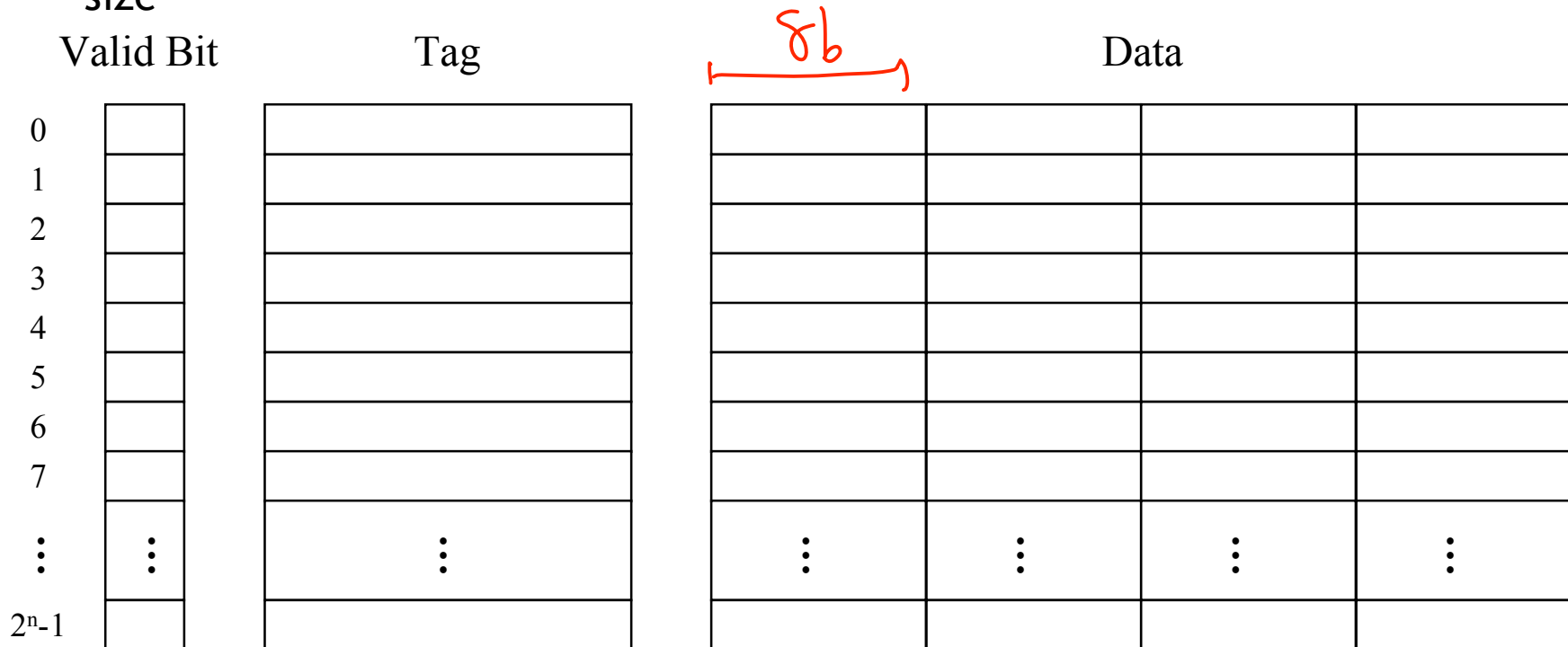


Total size: $\frac{25\text{b}}{\text{block}} \times 2^{16} \text{ blocks} = 200\text{KB}$

3x

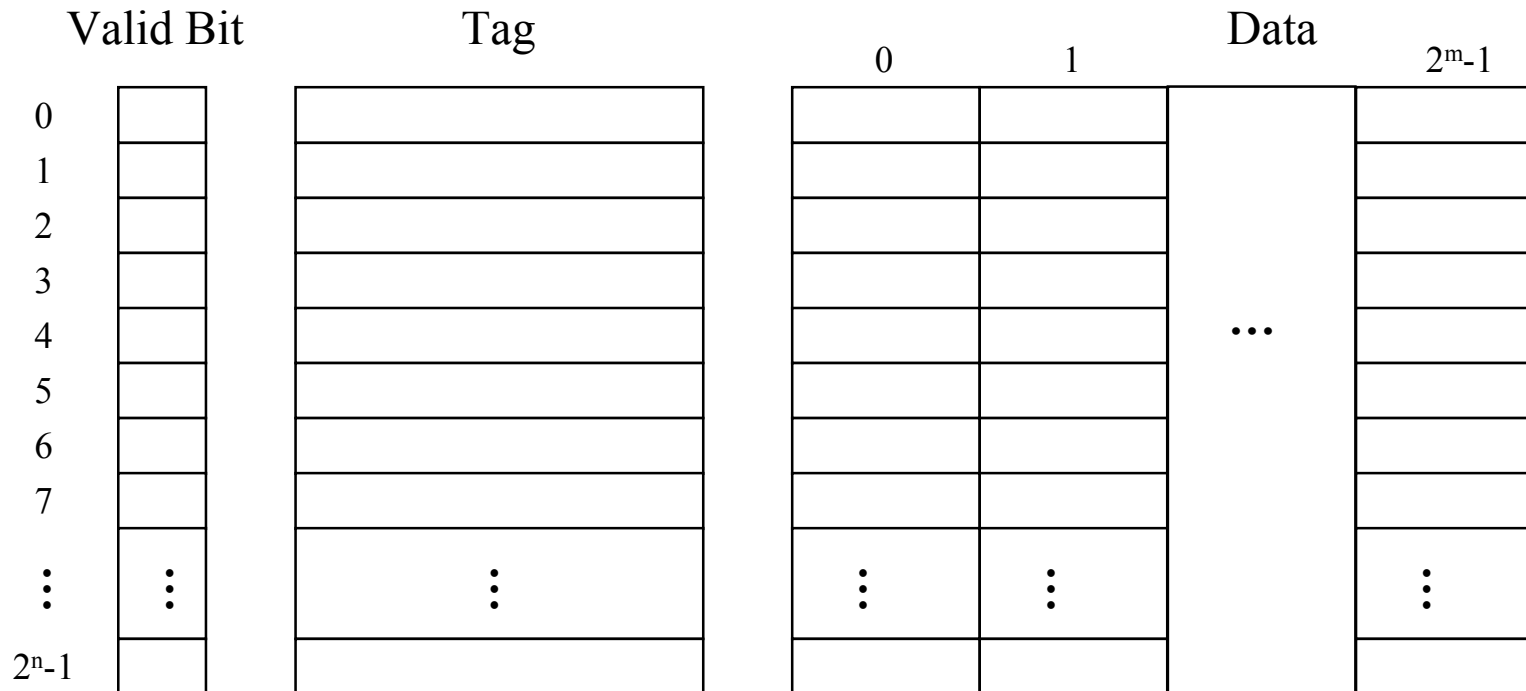
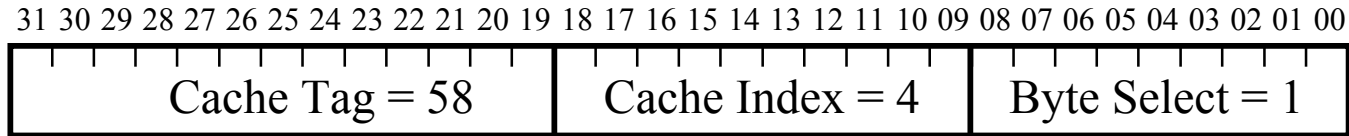
Cache Block Overhead

- Previous discussion assumed direct mapped cache 1 byte blocks
 - Uses temporal locality by holding on to previously used values
 - Does not take advantage of spatial locality
 - Significant area overhead for tag memory
- Take advantage of spatial locality & amortize tag memory via larger block size



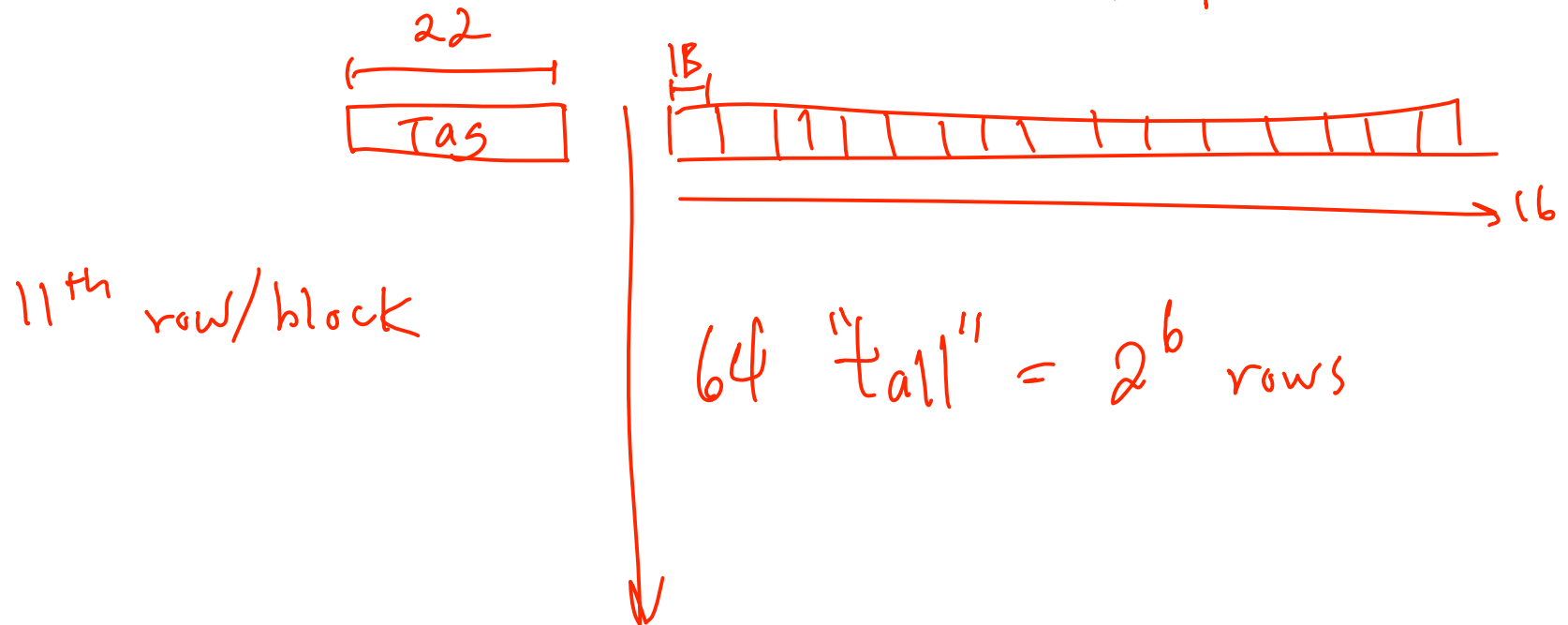
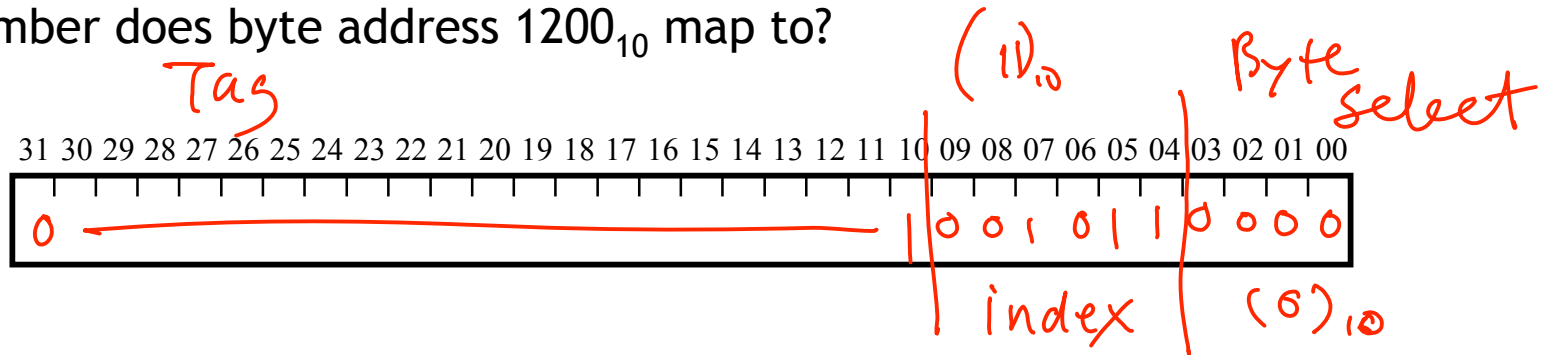
Cache Blocks

- Assume 2^n byte direct mapped cache with 2^m byte blocks



Cache Block Example

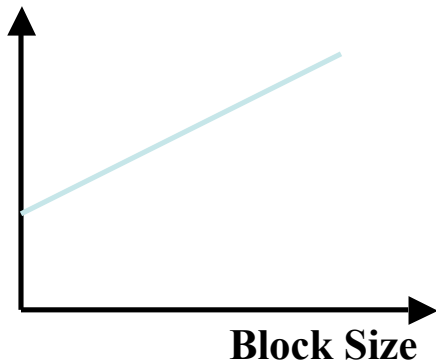
- Given a cache with 64 blocks and a block size of 16 bytes, what block number does byte address 1200_{10} map to?



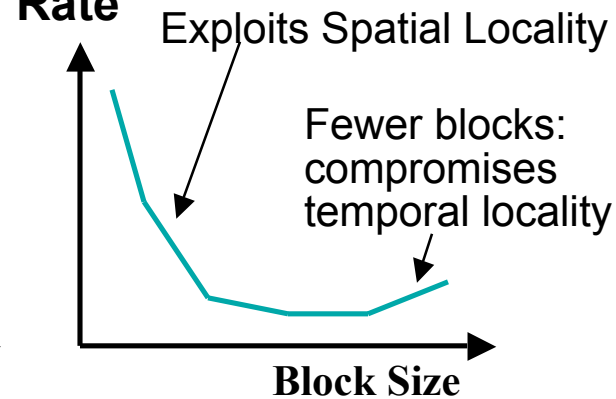
Block Size Tradeoff

- In general, larger block size take advantage of spatial locality **BUT**:
 - Larger block size means larger miss penalty:
 - Takes longer time to fill up the block
 - If block size is too big relative to cache size, miss rate will go up
 - Too few cache blocks

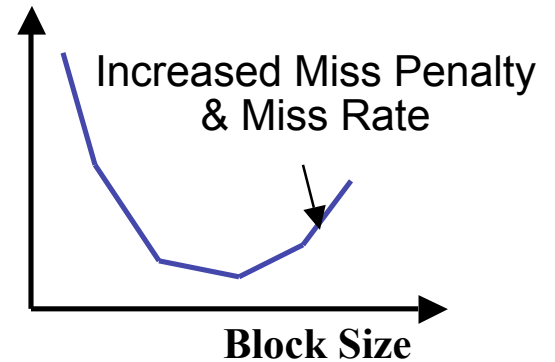
Miss Penalty



Miss Rate



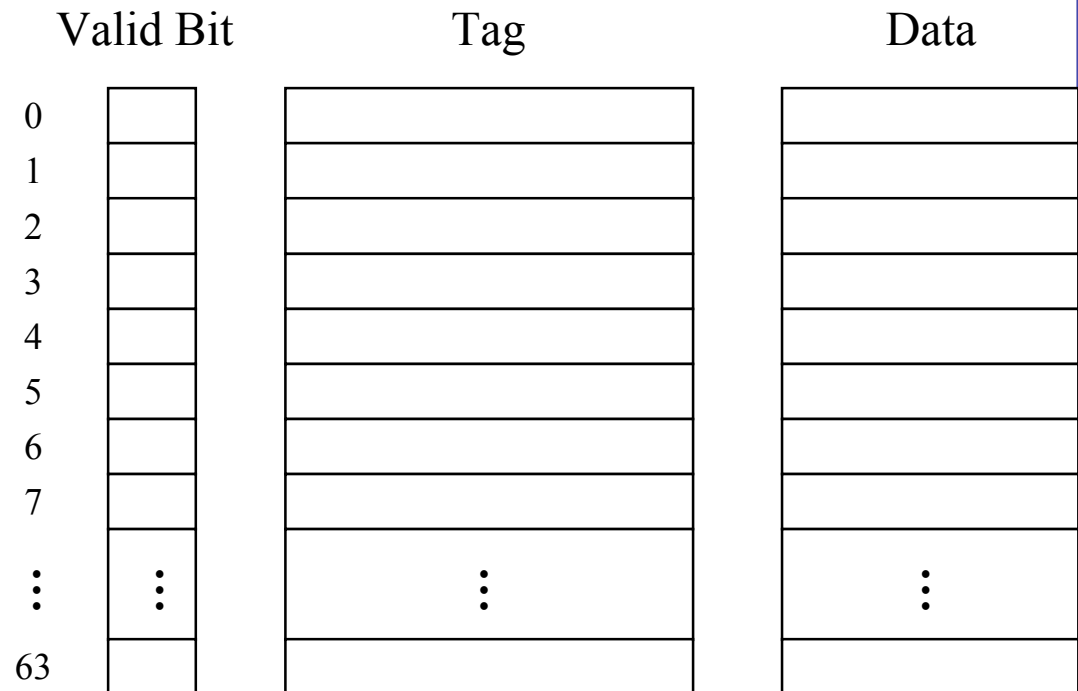
Average Access Time



Direct Mapped Cache Problems

- What if regularly used items happen to map to the same cache line?
- Ex. $\&(\text{sum}) = 0$, $\&(I) = 64$, cache is 64 bytes

```
int sum = 0;  
...  
for (int I=0; I!=N; I++) {  
    sum += I;  
}
```



- Thrashing - Continually loading into cache but evicting it before reuse

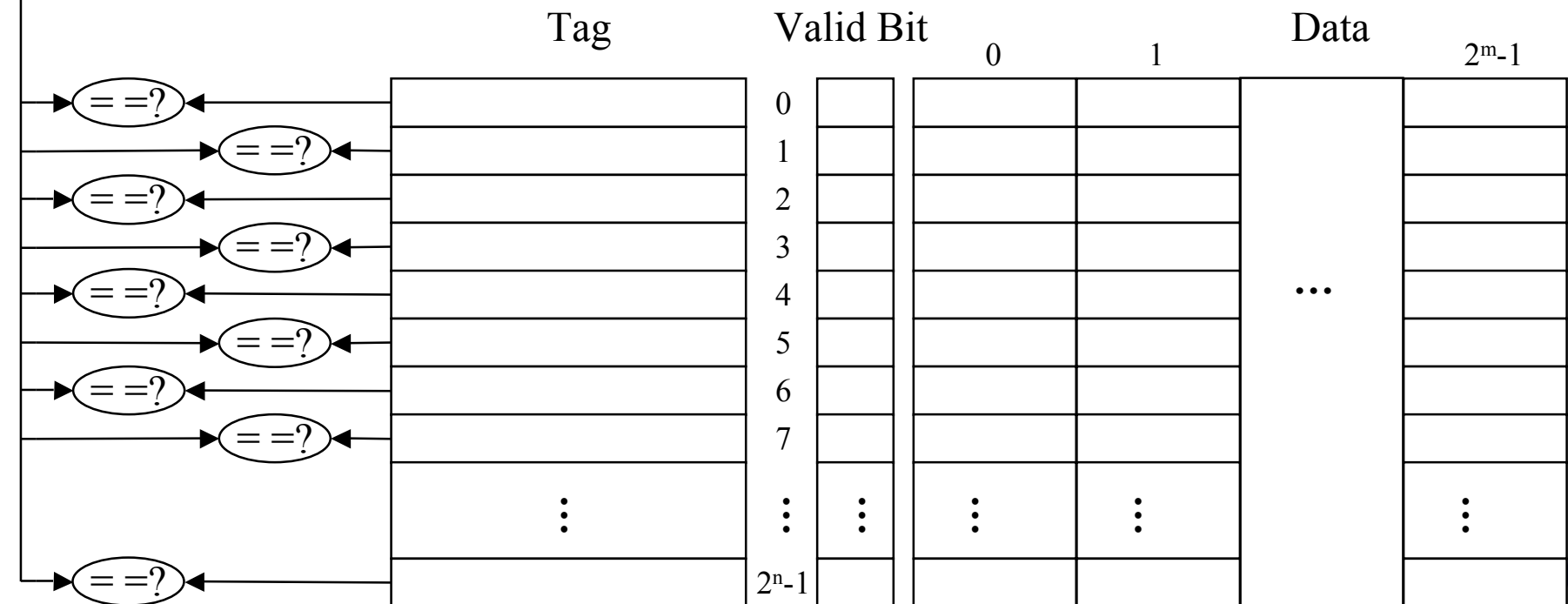
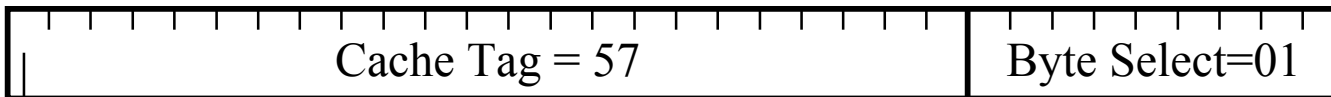
Cache Miss Types

- Several different types of misses (categorized based on problem/solution)
 - 3 C's of cache design
- Compulsory/Coldstart
 - First access to a block - basically unavoidable (though bigger blocks help)
 - For long-running programs this is a small fraction of misses
- Capacity
 - The block needed was in the cache, but unloaded because too many other accesses intervened.
 - Solution is to increase cache size (but bigger is slower, more expensive)
- Conflict
 - The block needed was in the cache, and there was enough room to hold it and all intervening accesses, but blocks mapped to the same location knocked it out.
- Solutions
 - Cache size
 - Associativity
- Invalidation
 - I/O or other processes invalidate the cache entry

Fully Associative Cache

- No cache index - blocks can be in any cache line

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00



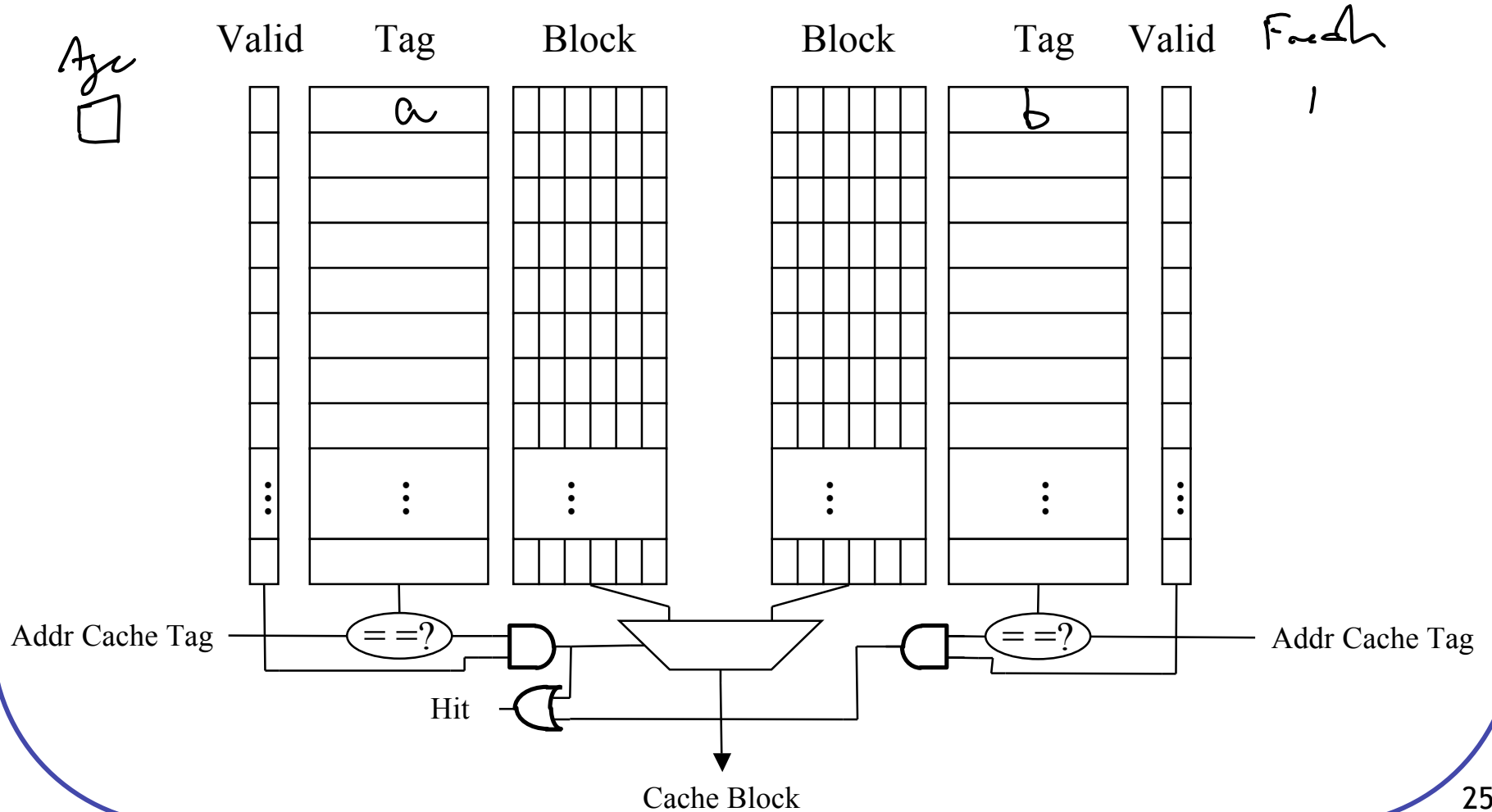
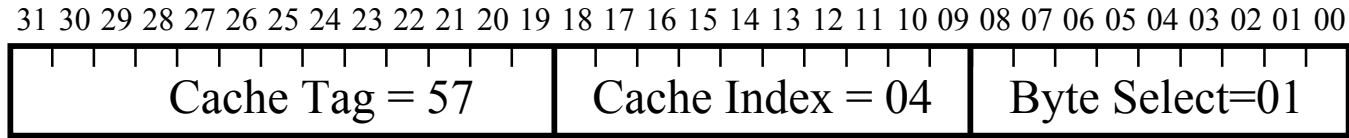
Fully Associative vs. Direct Mapped

N-way Set Associative

- N lines are assigned to each cache index
- ~ N direct mapped caches working in parallel
- Direct mapped = 1-way set associative
- Fully Associative = 2^N -way set associative (where 2^N is # of cache lines)

2-Way Set Associative Cache

- Cache index selects a "set", two tags compared in parallel



N-way vs. Other Caches

Cache Miss Comparison

- Fill in the blanks: Zero, Low, Medium, High, Same for all

	Direct Mapped	N-Way Set Associative	Fully Associative
Cache Size: Small, Medium, Big?			
Compulsory Miss:			
Capacity Miss			
Conflict Miss			
Invalidation Miss	Same	Same	Same

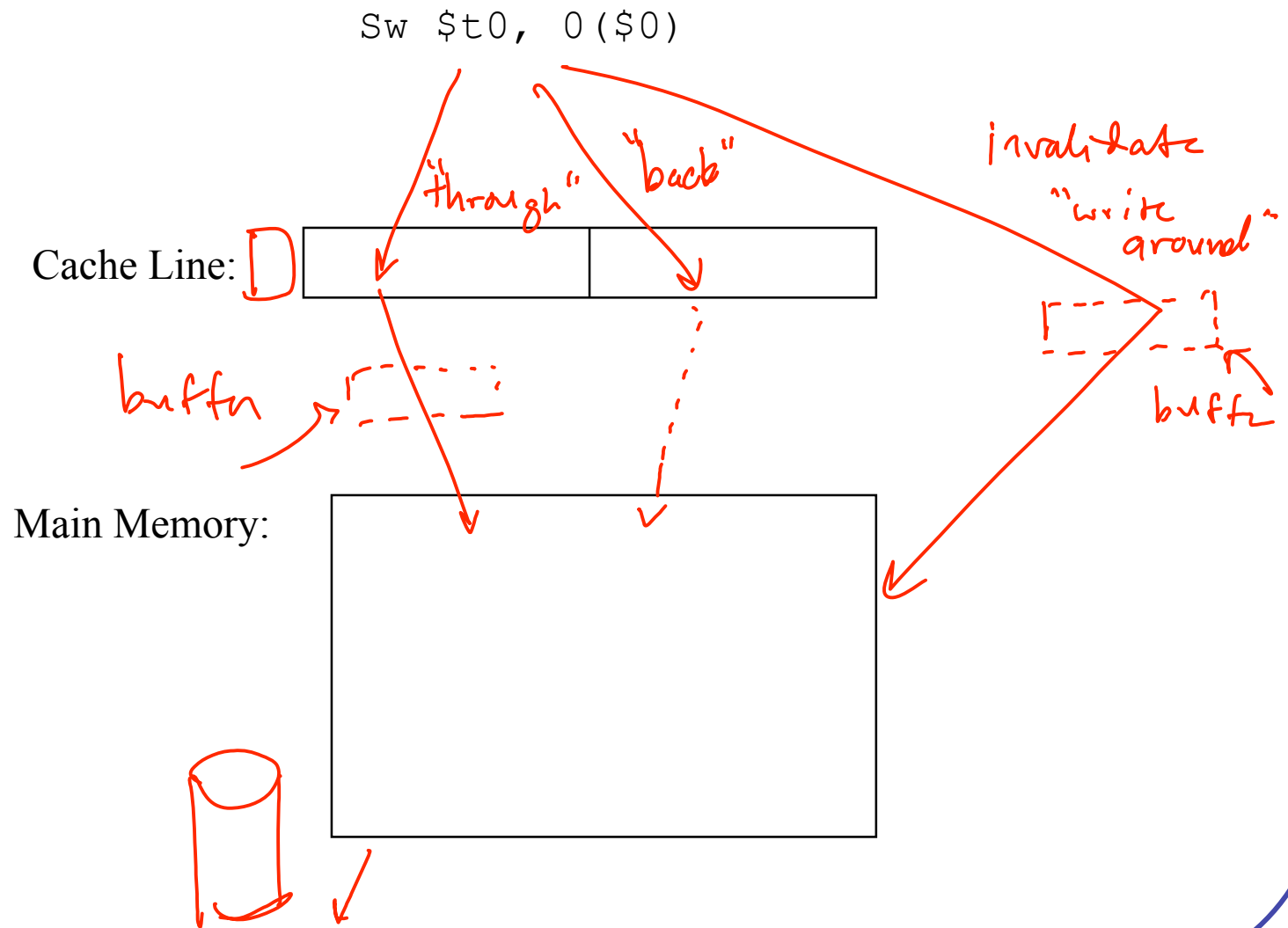
Complex Cache Miss Example

- 8-word cache, 2-word blocks. Determine types of misses (CAP, COLD, CONF).

Byte Addr	Block Addr	Direct Mapped	2-Way Assoc	Fully Assoc
0				
4				
8				
24				
56				
8				
24				
16				
0				
Total:				

Writing & Caches

- Direct-mapped cache with 2-word blocks, initially empty



Writing & Caches (cont.)

Replacement Methods

- If we need to load a new cache line, where does it go?
- Direct-mapped
- Set Associative
- Fully Associative

Replacement Strategies

- When needed, pick a location
- Approach #1: Random
 - Just arbitrarily pick from possible locations
- Approach #2: Least Recently Used (LRU)
 - Use temporal locality
 - Must track somehow - extra cache bits to indicate how recently used
- In practice, Random typically only 12% worse than LRU

Split Caches

- Instruction vs. Data accesses
 - How do the two compare in usage?
 - How many accesses/cycle do we need for our pipelined CPU?
- Typically split the caches into separate instruction, data caches
 - Higher bandwidth
 - Optimize to usage
 - Slightly higher miss rate because each cache is smaller.

Multi-level Caches

- Instead of just having an on-chip (L1) cache, an off-chip (L2) cache is helpful
- Ex. Consider instruction fetches only:
Base machine with CPI = 1.0 if all references hit the L1, 500 MHz
Main memory access delay of 200ns. L1 miss rate of 5%
How much faster would the machine be if we added a L2 which reduces the miss rate of L1 & L2 to 2%, but all L2 accesses (hits & misses) are 20ns, thus slowing down main memory accesses to 220ns.

Cache Summary
