Computer Organization and Architecture

Memory Hierarchy Basics / Cache

Jeongseob Ahn

Department of Software & Computer Engineering
Ajou University

Adapted slides from CIS501@Upenn

Review from Last Lecture

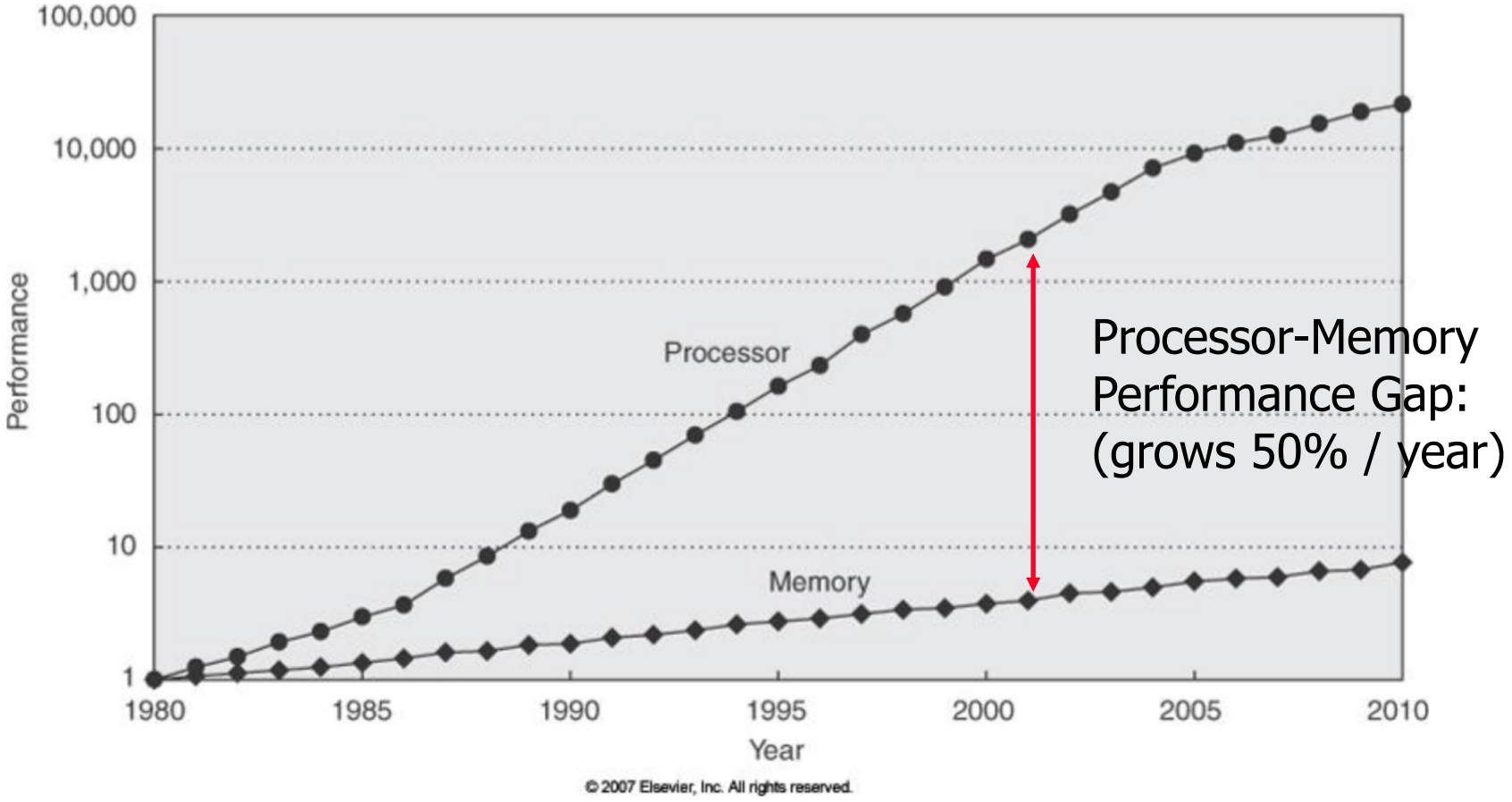
- Basic concept of pipelining: adding parallelism
- Universal solution for hazards → stalling pipeline
- Structural hazard → more resource bandwidth
- Data hazard → bypassing, compiler scheduling
- Control hazard → early branch resoultion, speculation

Outline

- Cache Basics: Organization and Performance
- Address Mapping: Direct-mapped vs. Associative Caches
- Reducing Hit Latencies
- Handling Misses and Reducing Miss Rates
- Write Policies

Processor-DRAM Memory Gap (Latency)

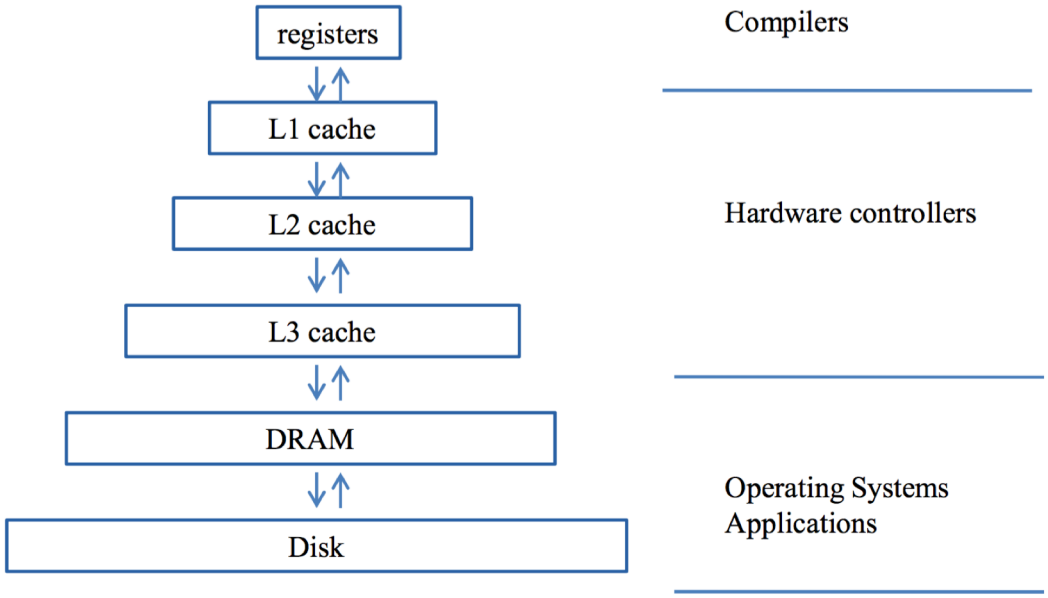
- Processor speed improvement: 60%/ year (2x/1.5year)
- Memory latency improvement: 9% / year (2x/10years)



Memory Hierarchy

- Trade-off speed vs. cost (or latency vs. capacity)
  - Small, fast, and expensive ←→ large, slow, and cheap components

- Exploit principle of locality (temporal and spatial)



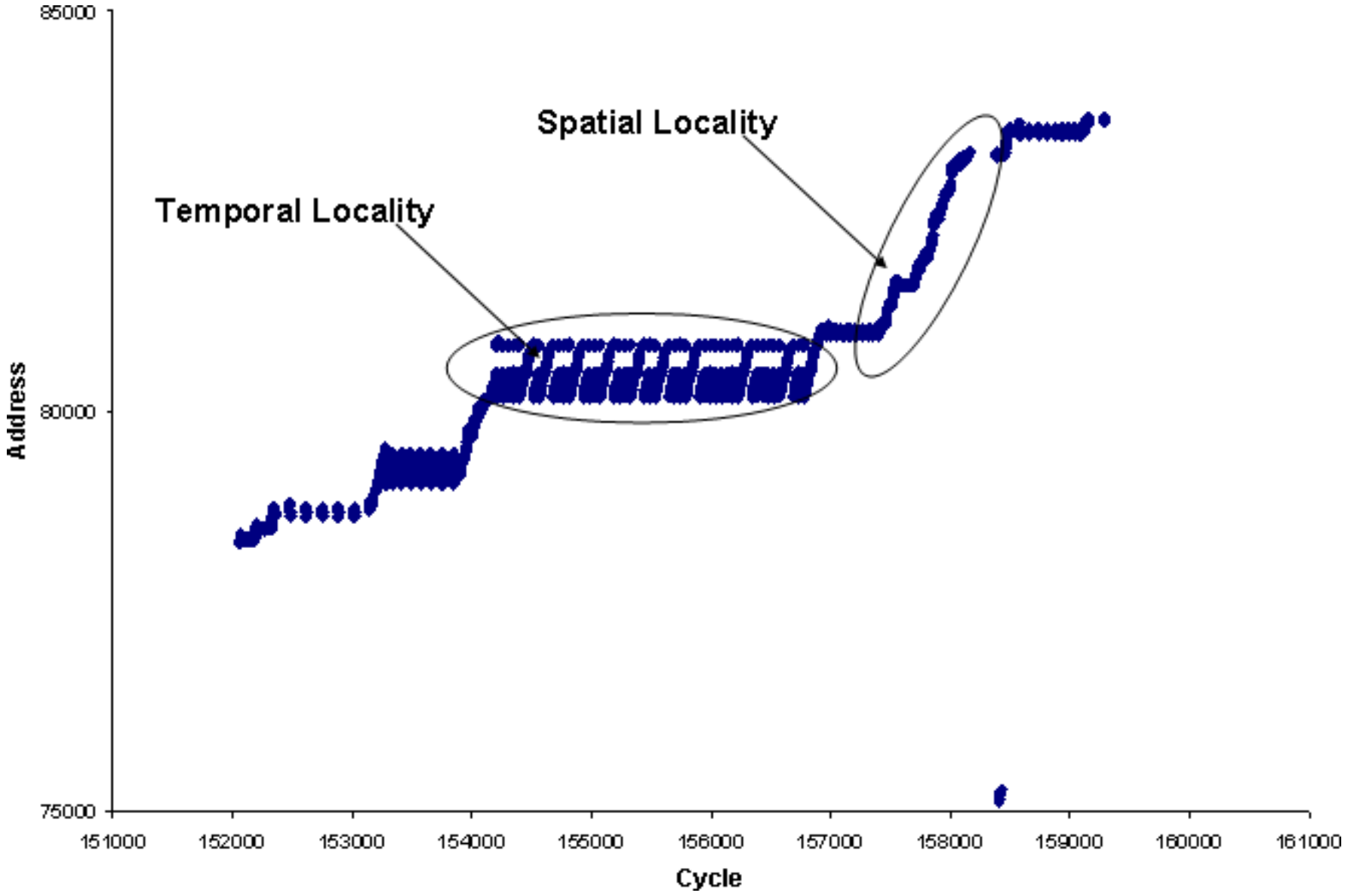
Principle of Locality

- Programs access a small proportion of their address space at any time

- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables

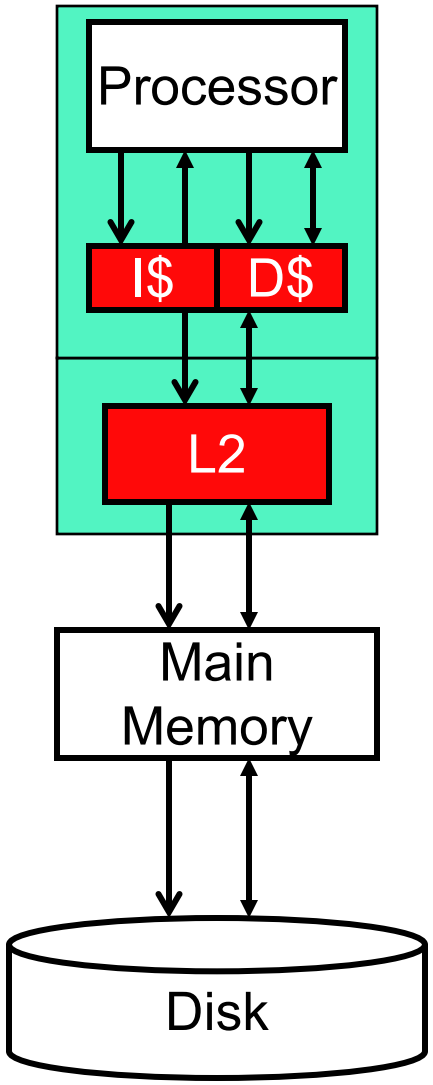
    An induction variable is a variable in a loop, whose value is a function of the loop iteration number $v = f(i)$

- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data



Reference: http://www.eetimes.com/

Caches



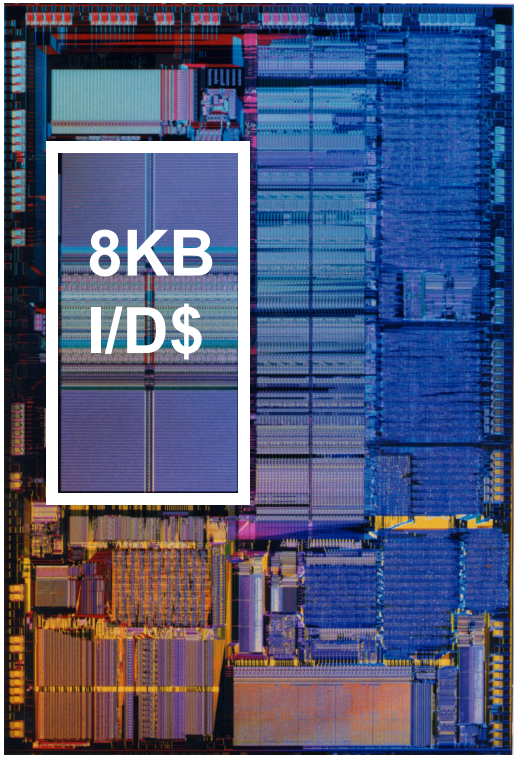
- "Cache": hardware managed
  - Hardware automatically retrieves missing data
  - Built from fast on-chip SRAM
  - In contrast to off-chip, DRAM "main memory"
- Cache ABCs (**associativity, block size, capacity,**)
- **Performance optimizations**
  - Prefetching & data restructuring
- **Handling writes**
  - Write-back vs. write-through
- **Memory hierarchy**
  - Smaller, faster, expensive → bigger, slower, cheaper

Cache Metrics

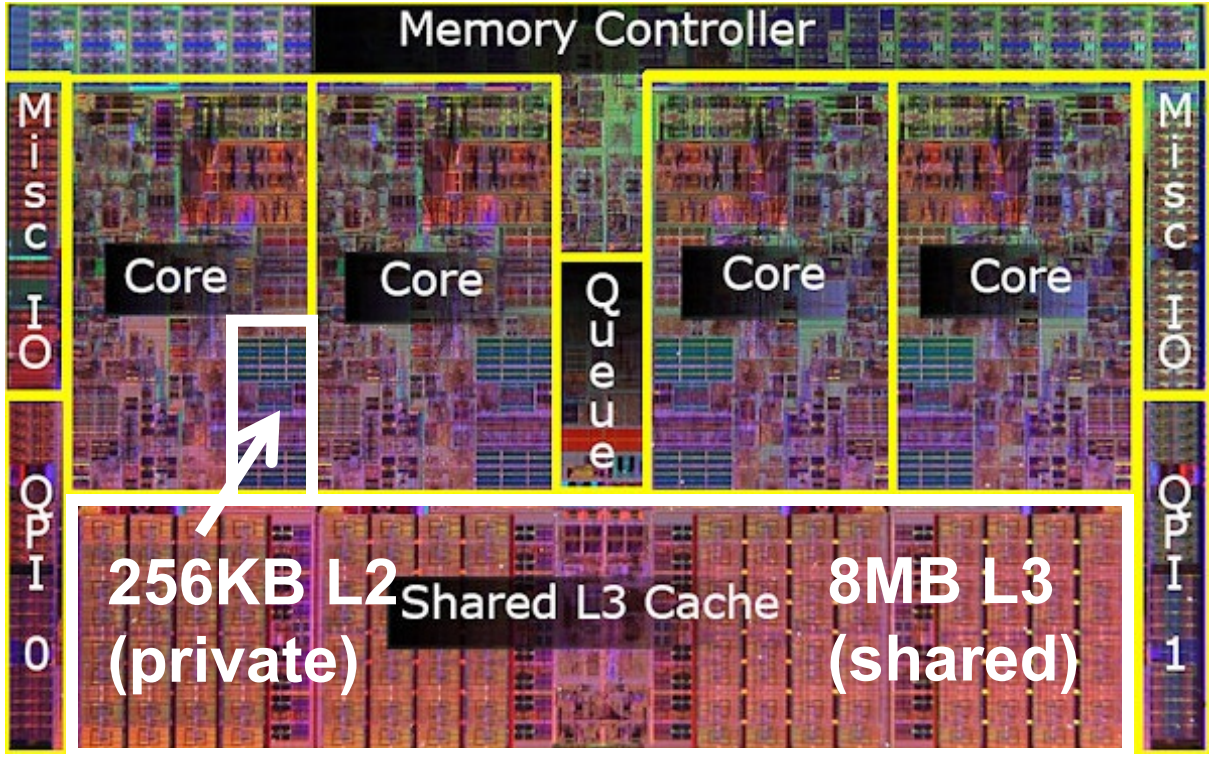
- Hit latency
  - Fixed for many L1(1-3 cycles) and L2 caches (8-20 cycles)
  - Some L2 and L3 caches may have variable latencies
- *Miss rate = cache misses / cache accesses*
- Miss latency
  - Time to fetch from the next level of cache or from the memory
- *Average access time = hit latency + miss rate x miss latency*

- Improving cache performance
  - Reduce hit latency → smaller caches
  - Reduce miss rates → larger caches
  - Reduce miss latency → better lower-level cache/memory

Evolution of Cache Hierarchies



Intel 486

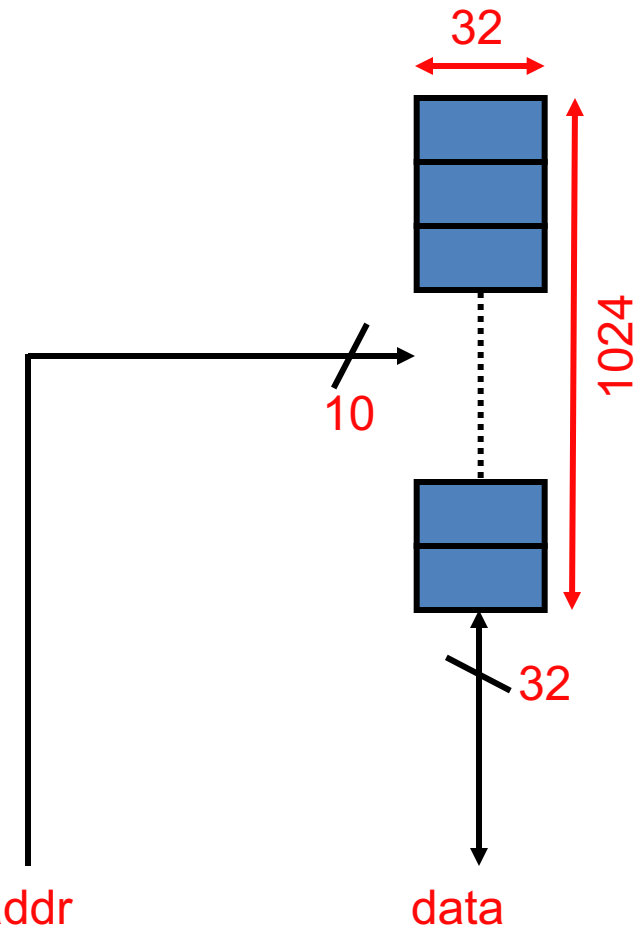


Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

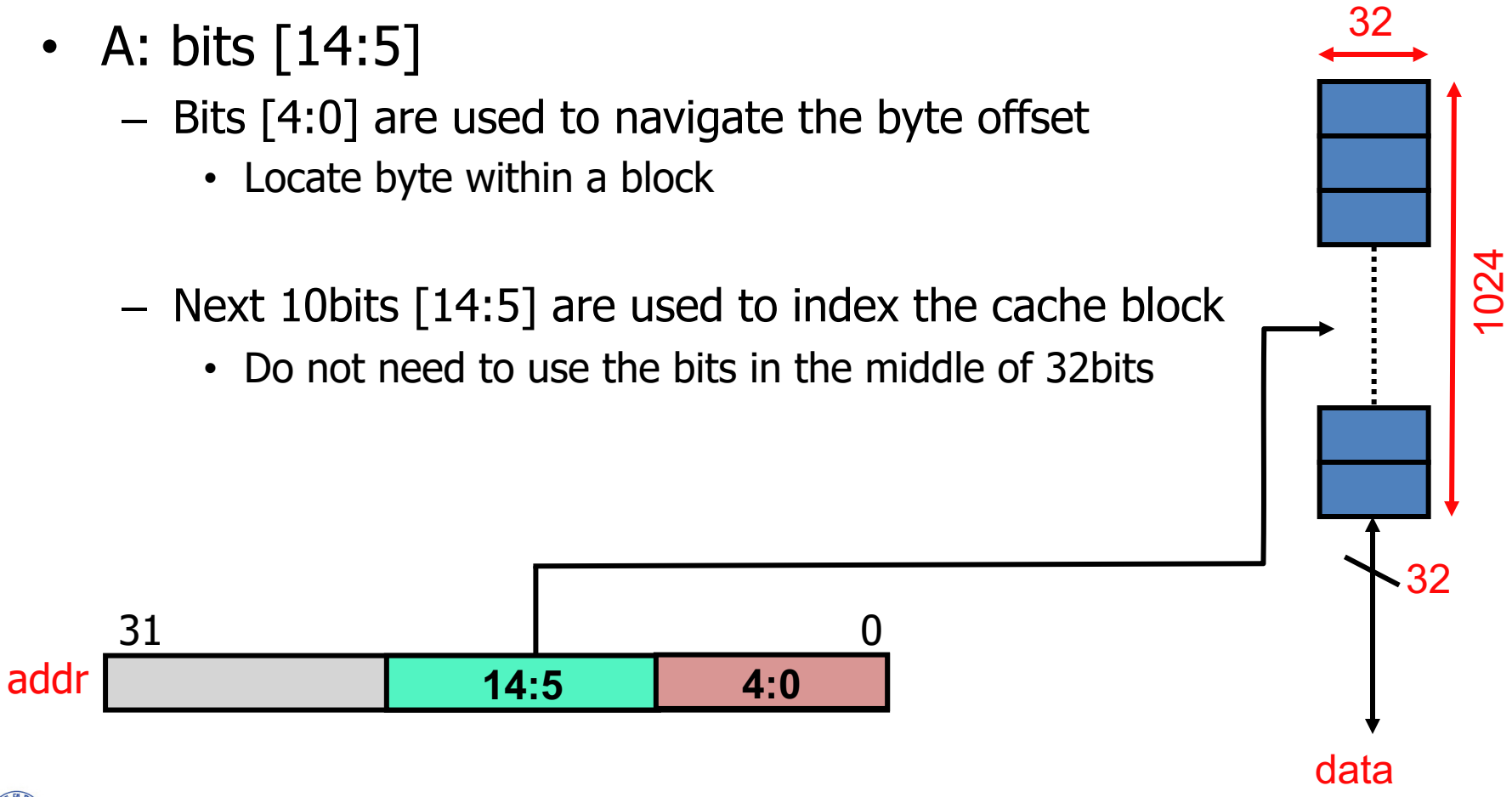
Hardware Cache Organization

- Cache is a hardware `hash table`
  - With one exception: data not always present
- Unit of data
  - Cache block (or cacheline)
  - 32byte, 64byte, …
- Cache organization
  - 32KB organized as 1K blocks of 32B each
  - Each block can hold a 32Byte
- How to find data from cache
  - Address mapping
- How to fill data to cache
  - Miss handling



Indexing: Looking Up A Block

- Q: Which 10 of the 32 bits address to use? (cache block: 32byte)
- A: bits [14:5]
  - Bits [4:0] are used to navigate the byte offset
    - Locate byte within a block
  - Next 10bits [14:5] are used to index the cache block
    - Do not need to use the bits in the middle of 32bits

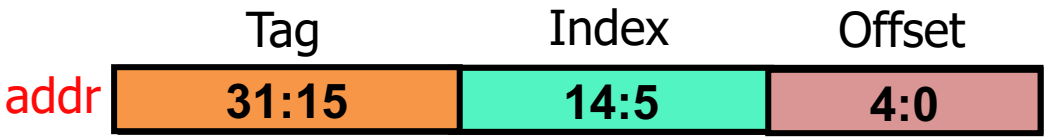


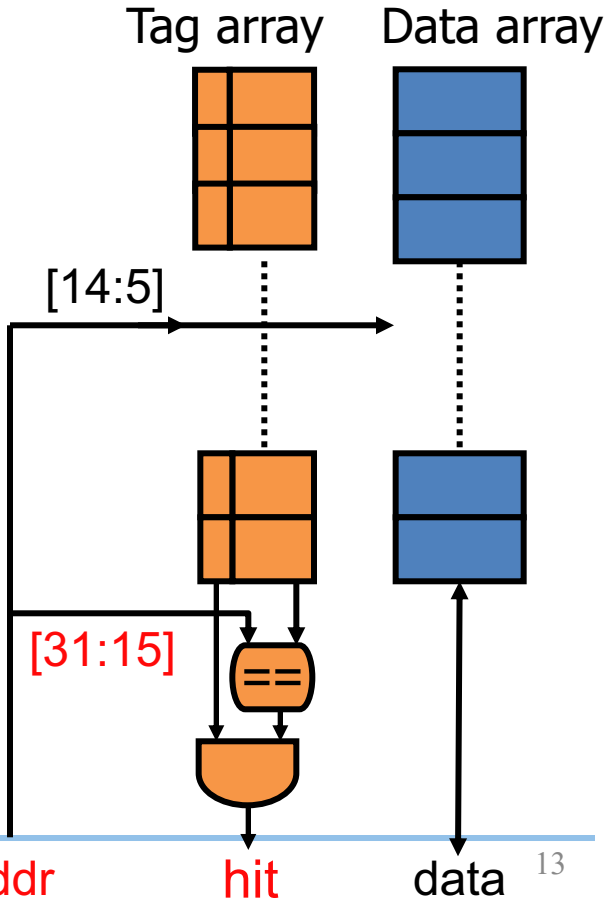
Tag Matching: Knowing that You Found It

- 2^{17} blocks map to each cache row
  - How to know which if any is currently there?
  - Tag each cache block with remaining address bits [31:15]

- Build separate and parallel **tag array**
  - 1K by 18-bit SRAM
  - 17-bit Tag + 1 valid bit

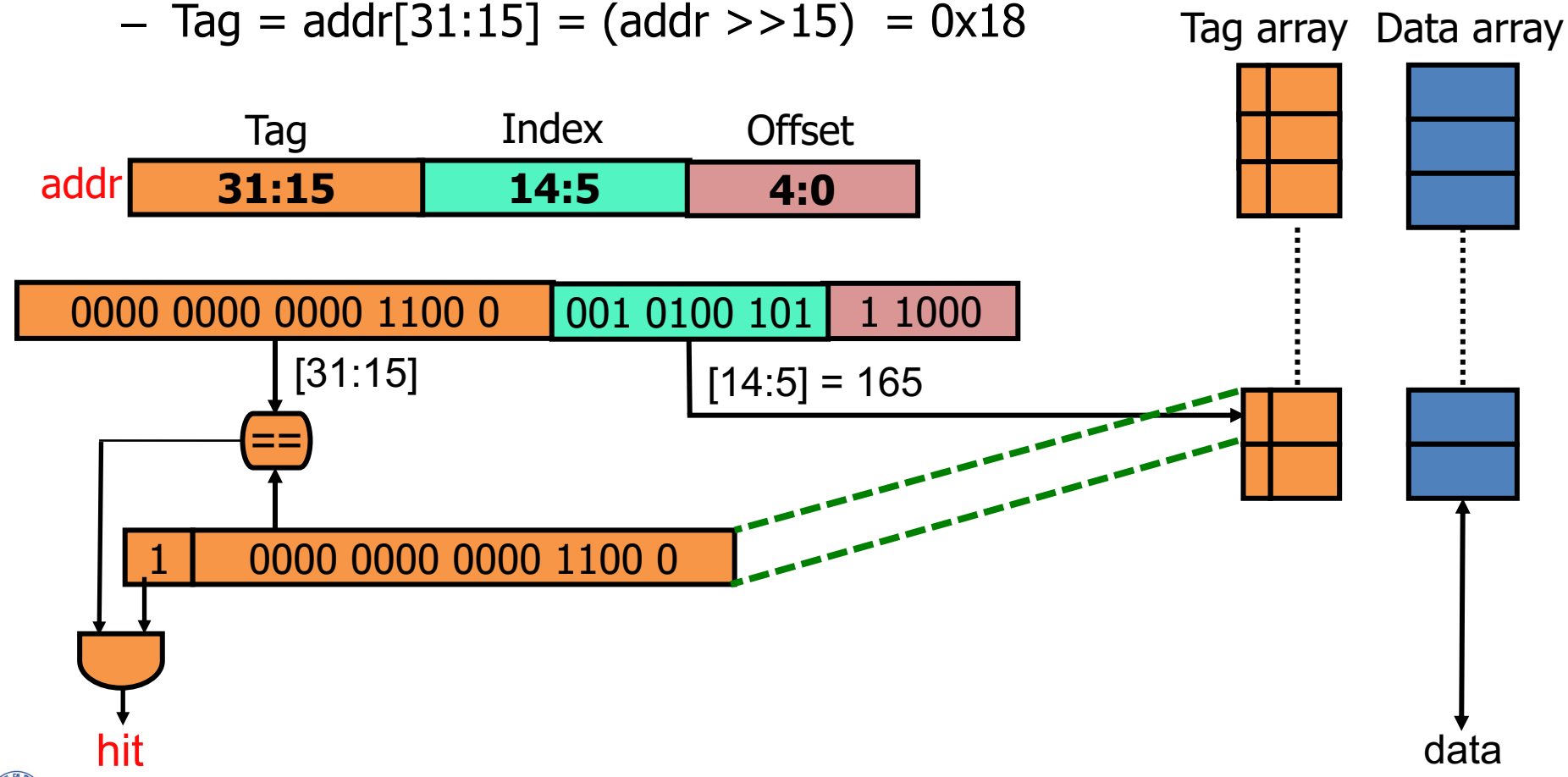
- Lookup algorithm
  - Read tag indicated by index bits
  - If tag matches & valid bit set:
    - Then: hit → data is good
    - Else: Miss→ data is garbage, wait…





A Concrete Example

- Lookup address 0x000c14B8
  - Index = addr[14:5] = (addr >> 5) & 0x3ff = 0xA5
  - Tag = addr[31:15] = (addr >>15) = 0x18



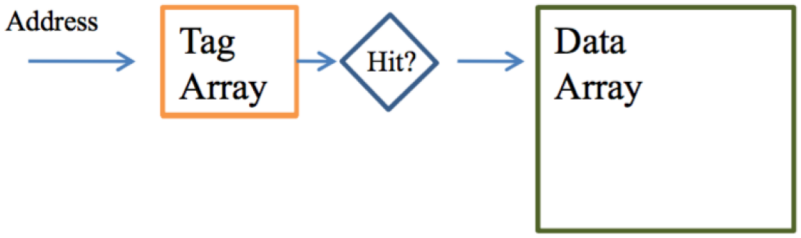
Calculating Tag Overhead

- "32KB cache" means cache holds 32KB of data
  - Called capacity
  - Tag storage is considered overhead

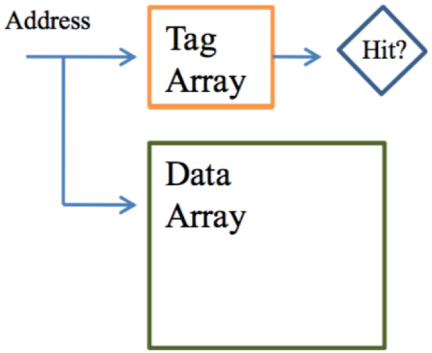
- Tag overhead of 32KB cache with 1024 32B blocks
  - 17bit tag + 1bit valid = 18bit
  - 18bit * 1024 = 18Kb tags = 2.25KB
  - 7% overhead

Serial and Parallel Tag/Data Access

- Serial tag and data array access
  - Access data array only for hits
  - Long hit latency (-): tag access time + data access time
  - Saving power (+)



- Parallel tag and data array access
  - Access data array for all accesses
  - Short hit latency (+)
  - Wasting power for misses (-)

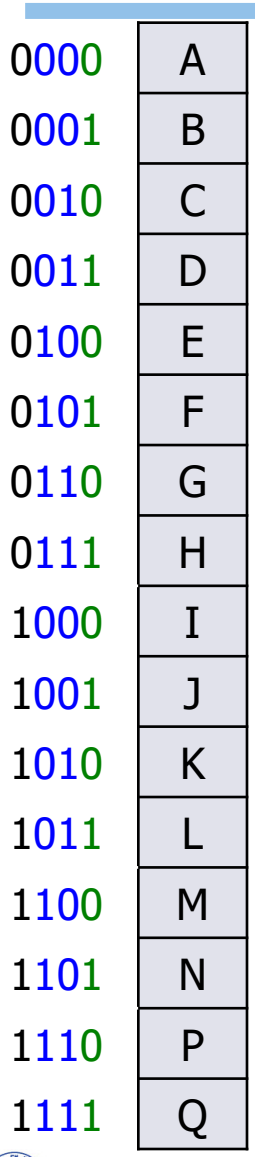


Handling a Cache Miss

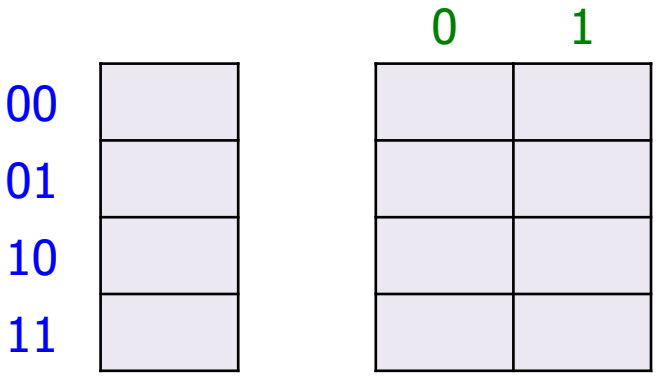
- What if requested data isn't in the cache?
  - How does it get in there?

- Cache controller: finite state machine
  - Remember miss address
  - Accesses next level of memory (L1 → L2 → L3 → DRAM)
  - Waits for response
  - Writes data/tag into proper locations

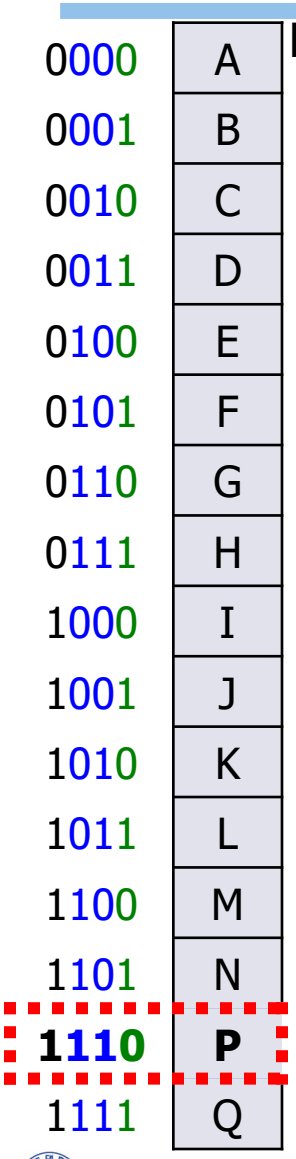
Example: 4-bit Address, 8B Cache, 2B Blocks



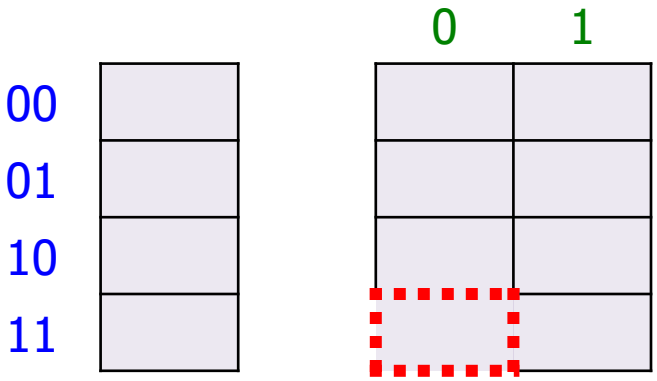

```
Load: 1110
```



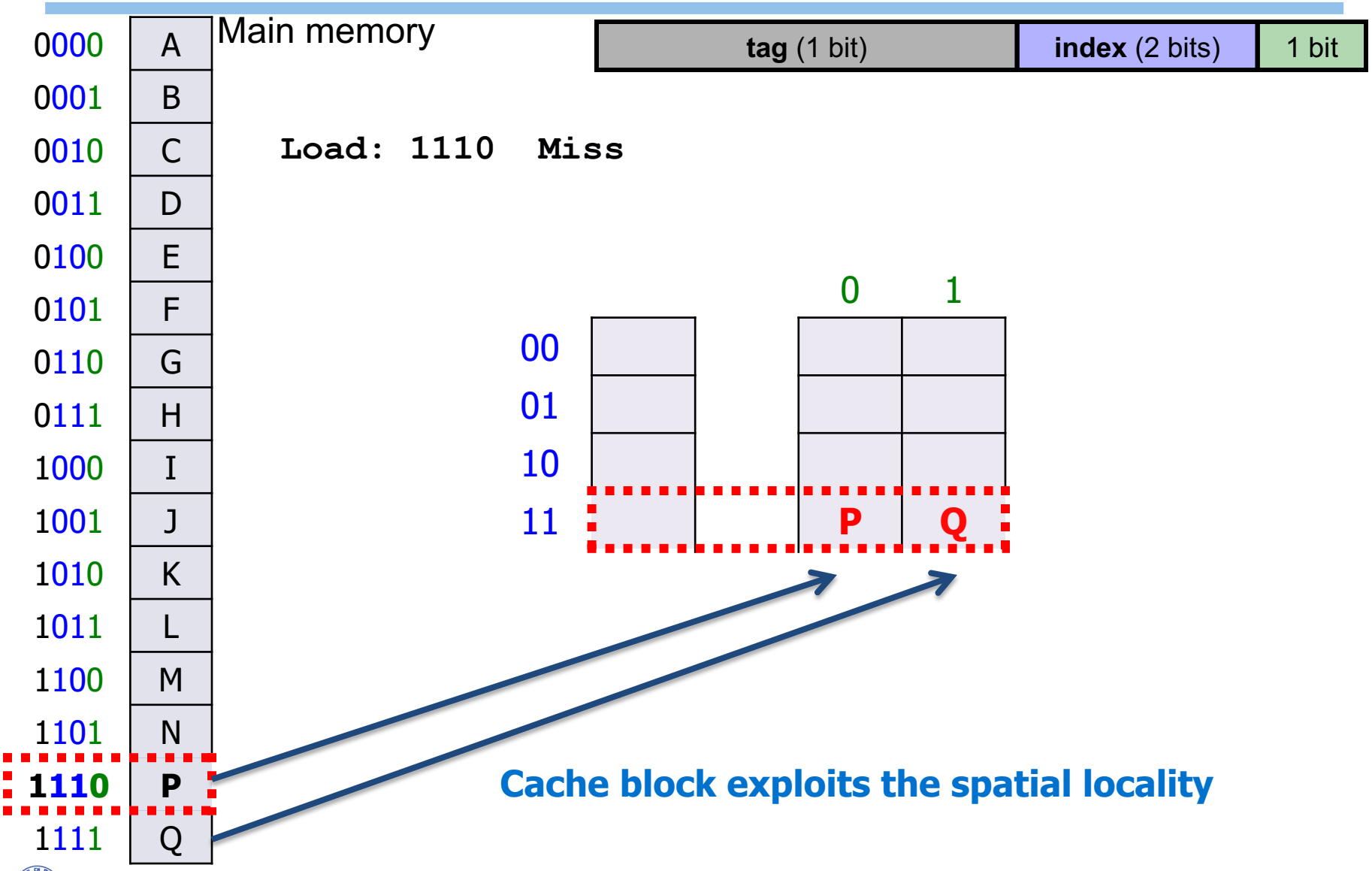
Example: 4-bit Address, 8B Cache, 2B Blocks



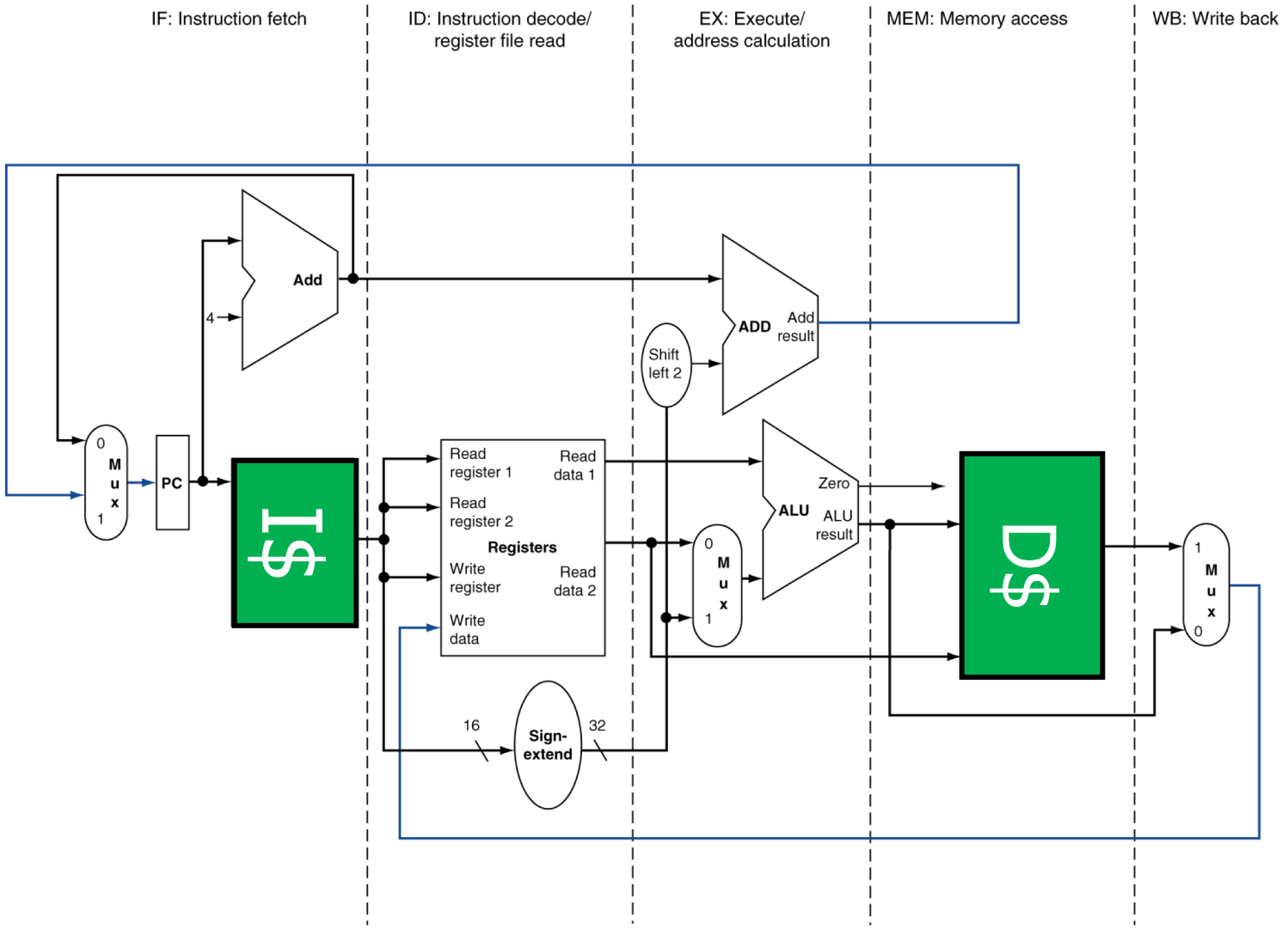
Load: 1110 Miss



Example: 4-bit Address, 8B Cache, 2B Blocks



Cache Misses and Pipeline Stalls



Split vs. Unified Caches

- Split I$/D$: instructions and data in difference caches
  - To minimize structural hazards and t_{hit}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ to fetch multiple instructions, no writes

- Unified L2, L3: instruction and data together
  - To minimize %miss
  - Fewer capacity misses: unused instruction capacity can be used for data
  - More conflict misses: instruction/data conflicts
    - A much smaller effect in large caches
  - Instruction/data structural hazards are rare: simultaneous I$/D$ miss

Replacement Policy

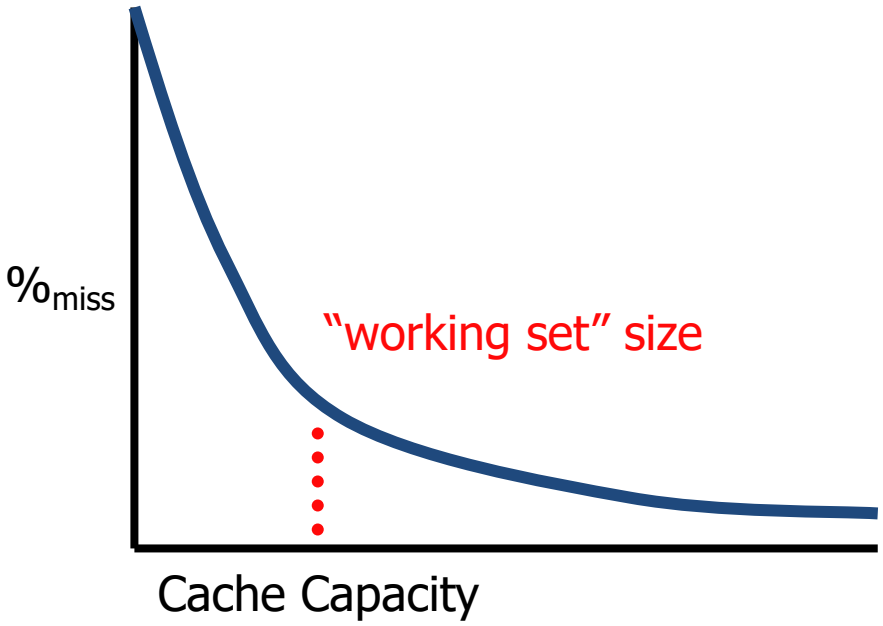
- If an empty space available
  - Use the empty block space

- If not, evict an existing one
  - Random
  - FIFO (first-in first-out)
  - LRU (least recently used)
  - Pseudo LRU: pure LRU is hard to implement with high associative caches

Try to achieve the temporal locality in general!

Capacity and Performance

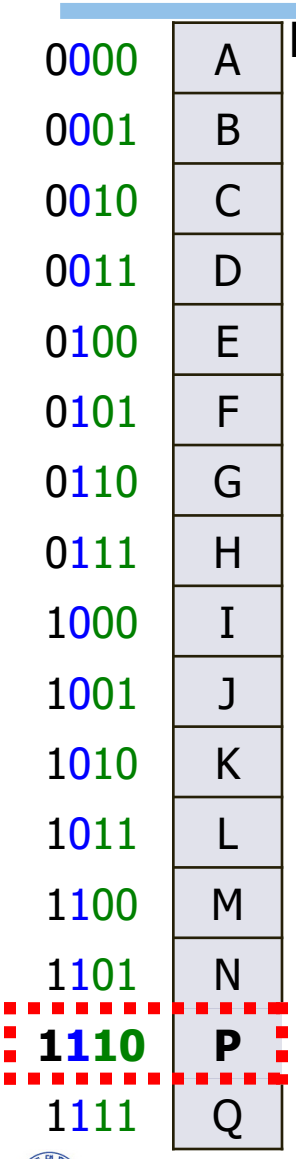
- Simplest way to reduce $\%_{miss}$: increase capacity
  - (+) Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
  - (-) However t_{hit} increases
    - Latency grows with cache size



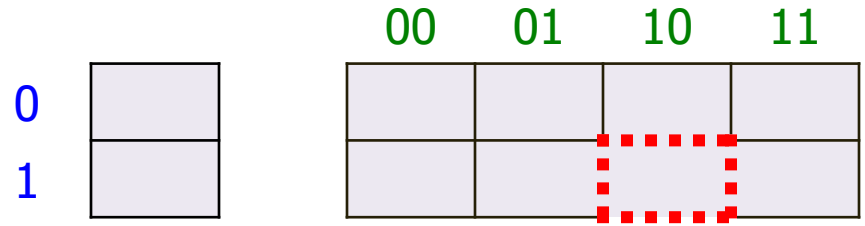
Effect of Cache Block Size

- Large blocks
  - Prefetching effects (+): exploit spatial locality (reduce compulsory misses)
  - Capacity waste (-): if small portion of blocks are actually needed
  - Miss latency (?): depend on bus width from next-level memory hierarchy
  - Minor reduction in tag storage overhead (+)

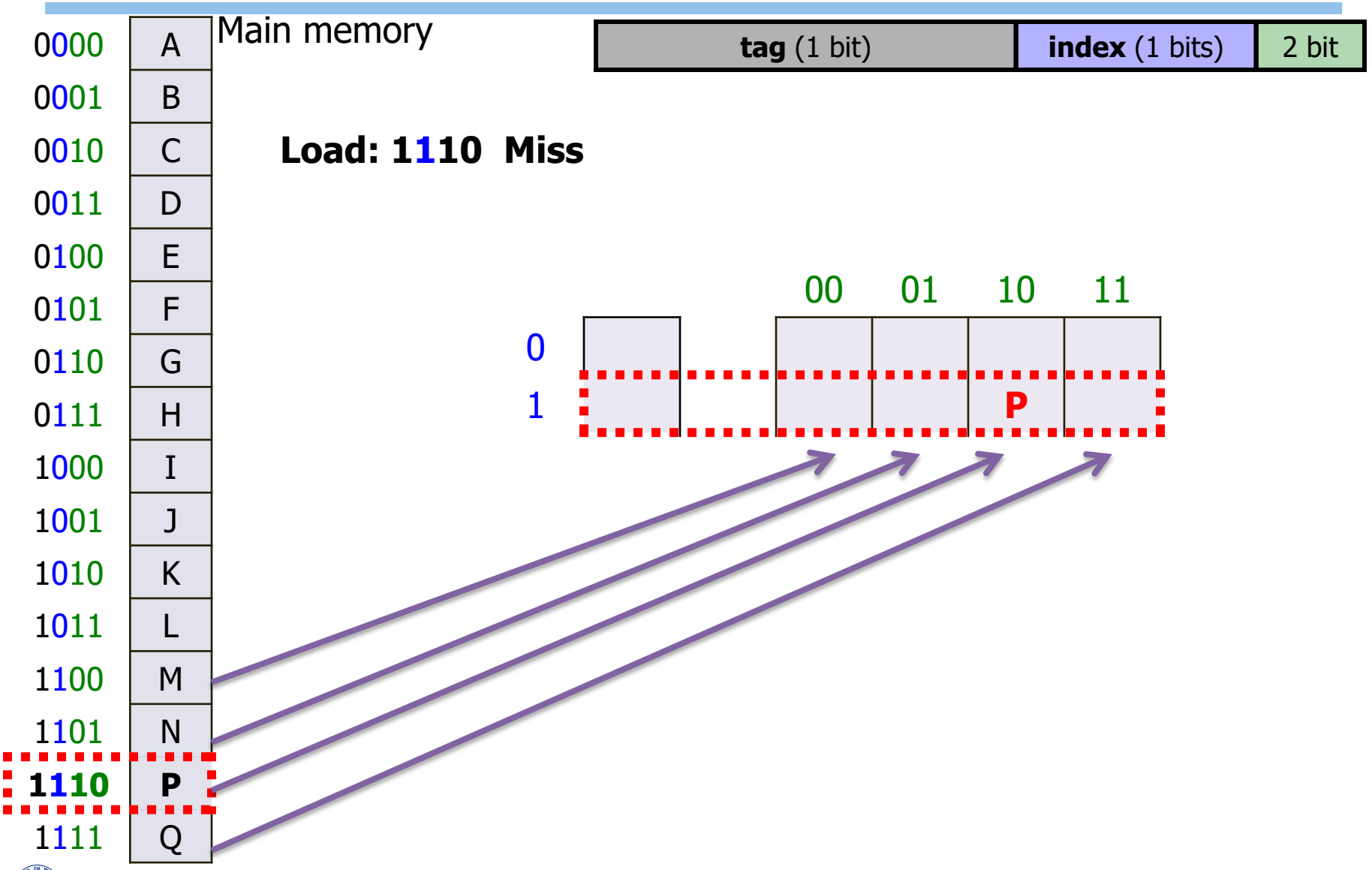
4-bit Address, 8B Cache, 4B Blocks



Load: 1110 Miss



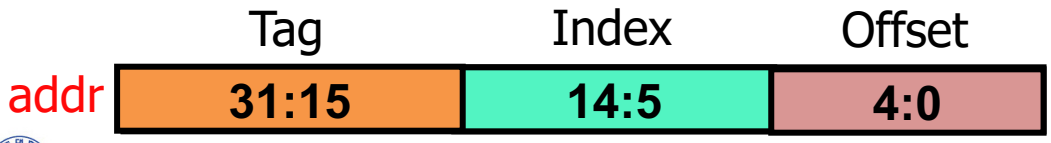
4-bit Address, 8B Cache, 4B Blocks

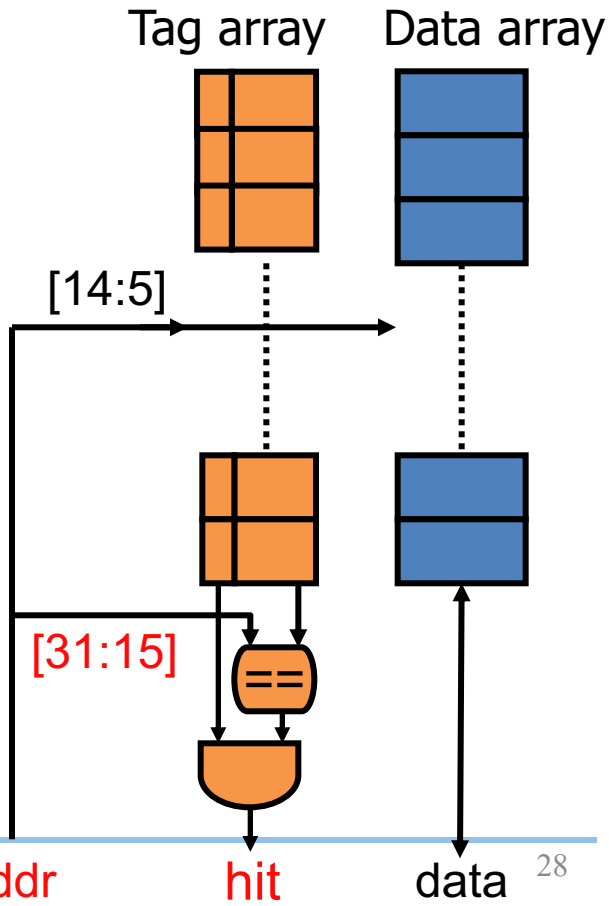


Cache Conflicts

- Consider two frequently-accessed variables…
- What if their addresses have the same "index" bits?
  - Such addresses "conflict" in the cache
  - Can't hold both in the cache at once…
  - Can result in lots of misses (bad!)

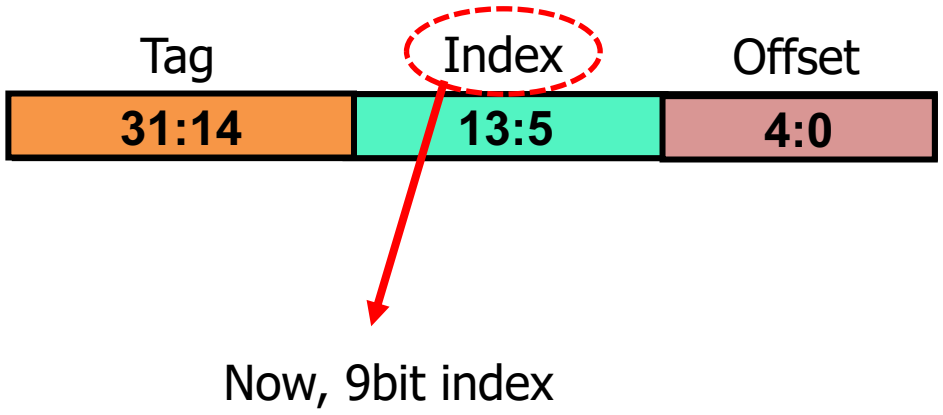
- Conflicts increase cache miss rate
  - Worse, result in non-robust performance
  - Small program change → changes memory layout → changes cache mapping of variables → dramatically increase/decrease conflicts

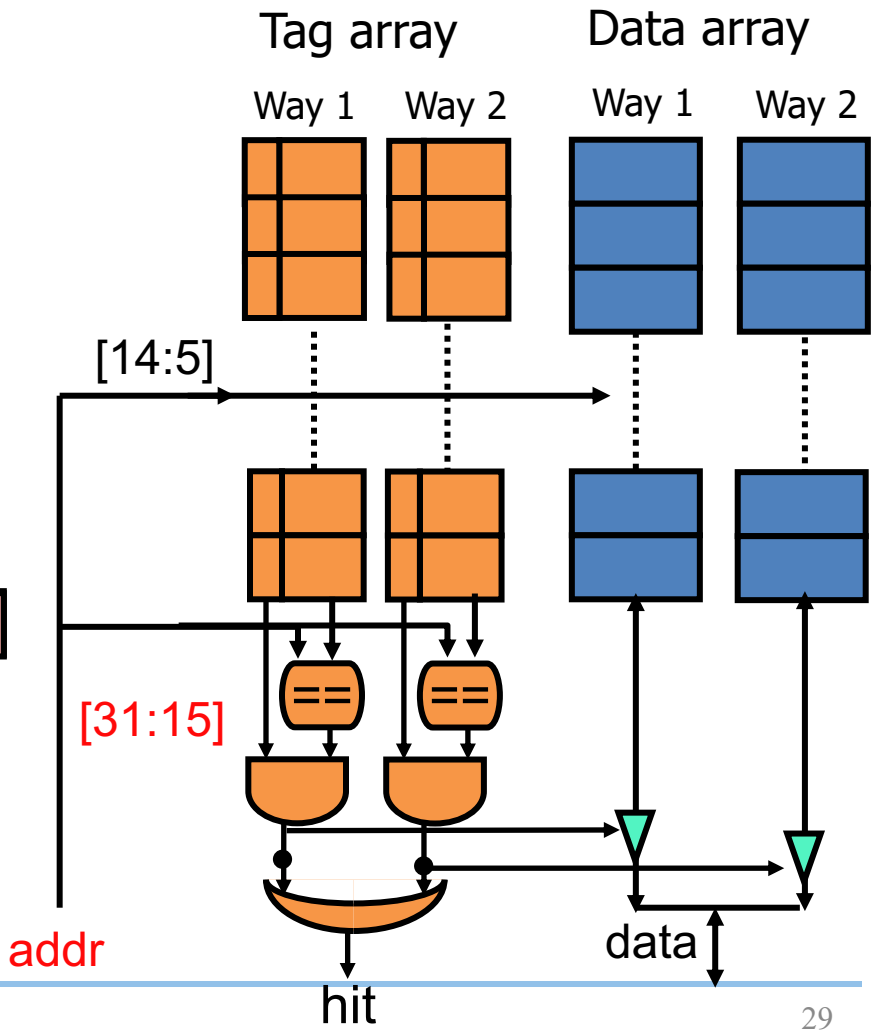




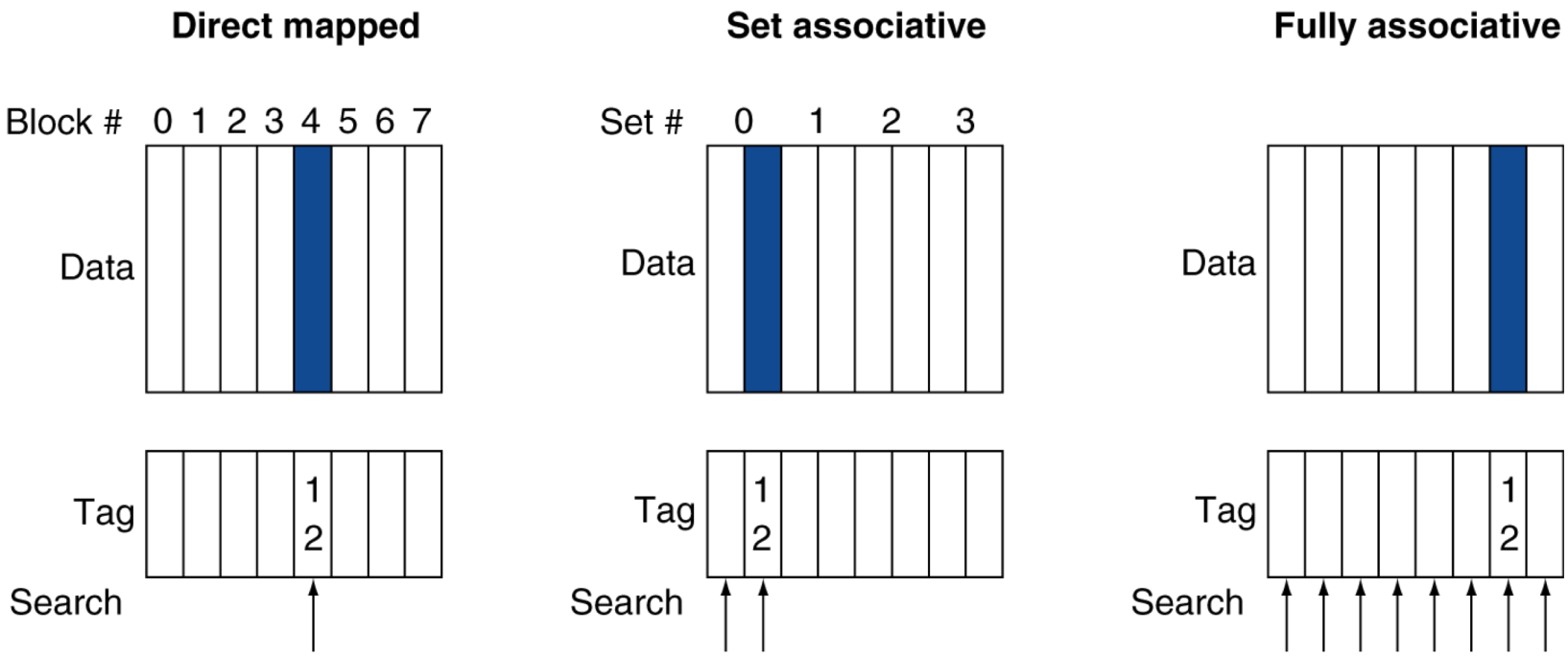
Associativity

- Set-associativity: *Two more possible locations* for each address
  - \+ Reduces conflicts
  - \- Increases latency$_{hit}$





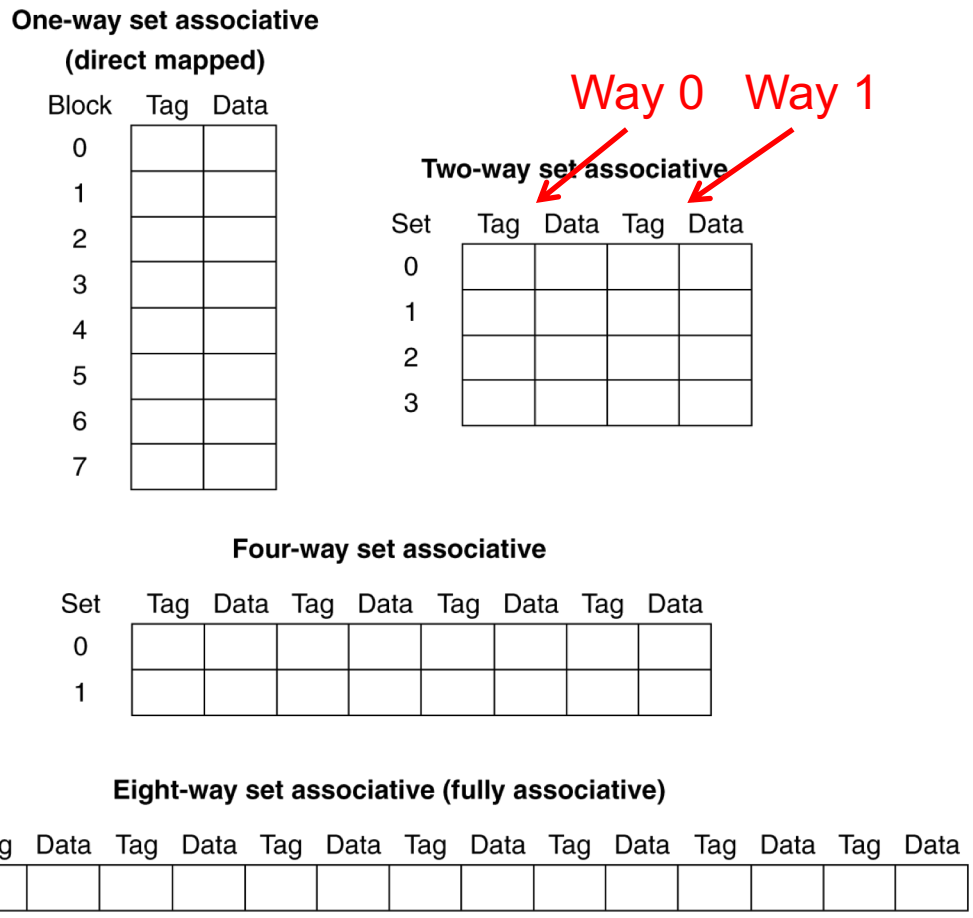
Associative Cache Example



2-way set associative

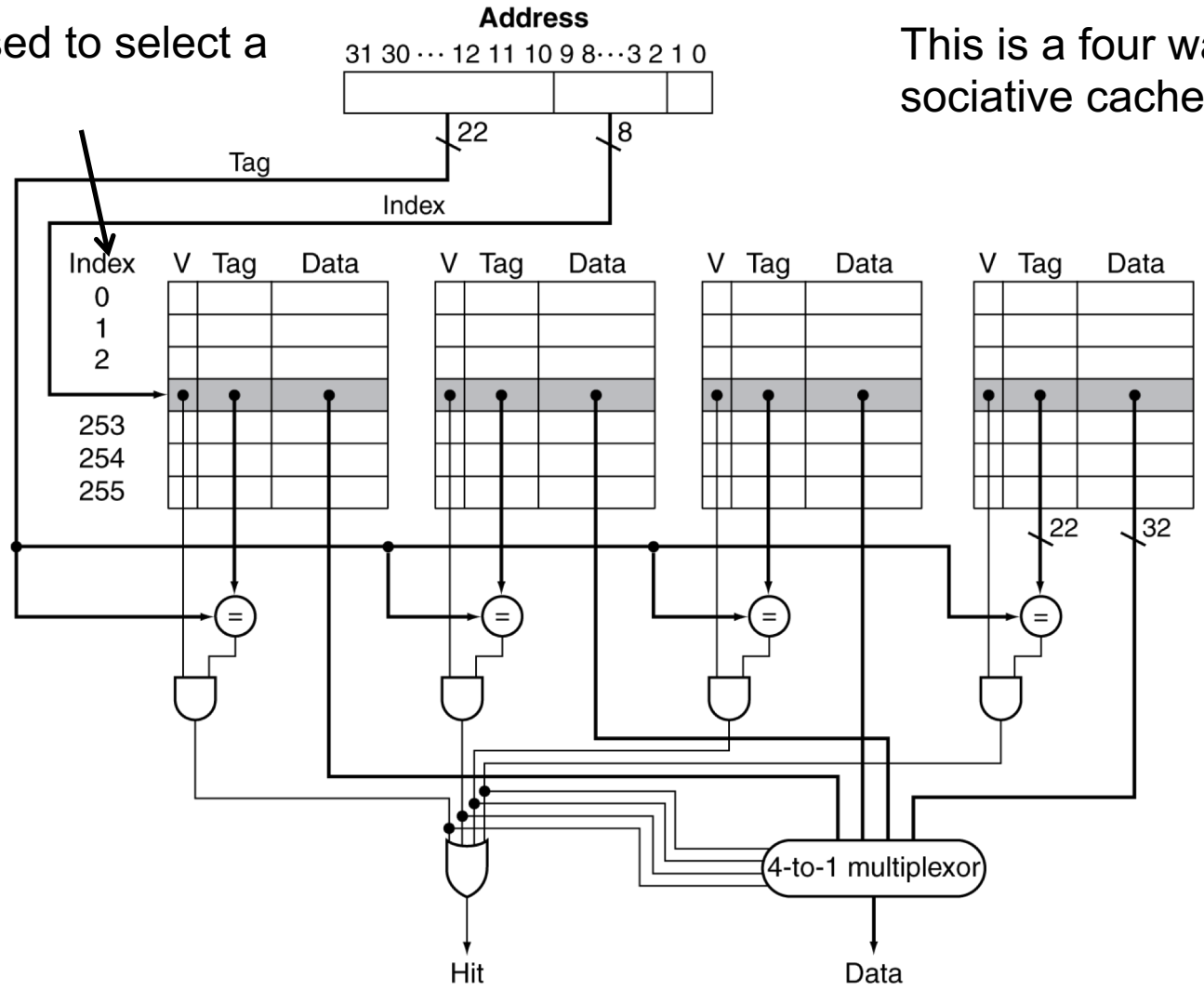
Spectrum of Associativity

- For a cache with 8 entries



Set Associative Cache Organization

Index is used to select a set



This is a four way set as sociative cache

4-bit Address, 8B Cache, 2B Blocks, 2-way

0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	P
1111	Q

Main memory

	Way 0			LRU	Way 1		
		Data				Data	
Set	Tag	0	1		Tag	0	1
0	00	A	B	0	01	E	F
1	00	C	D	1	01	G	H

4-bit Address, 8B Cache, 2B Blocks, 2-way

0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	**P**
1111	Q

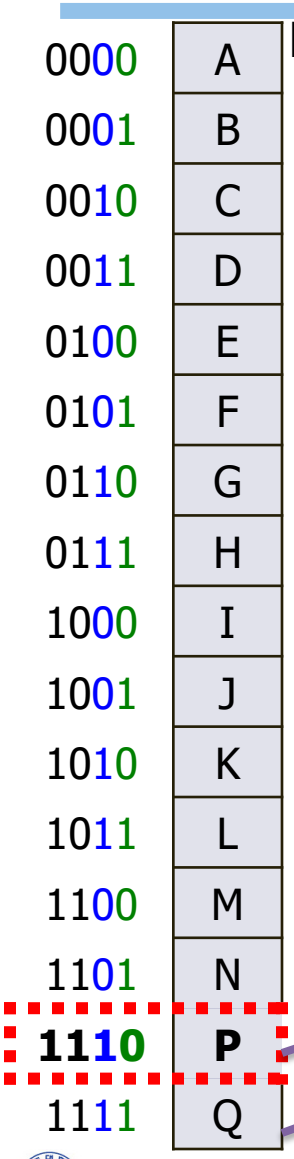
Main memory

tag (2 bit)	**index** (1 bits)	1 bit

Load: 1110 Miss

	Way 0			LRU	Way 1		
		Data				Data	
Set	**Tag**	0	1		**Tag**	0	1
0	00	A	B	0	01	E	F
1	00	C	D	1	01	G	H

4-bit Address, 8B Cache, 2B Blocks, 2-way



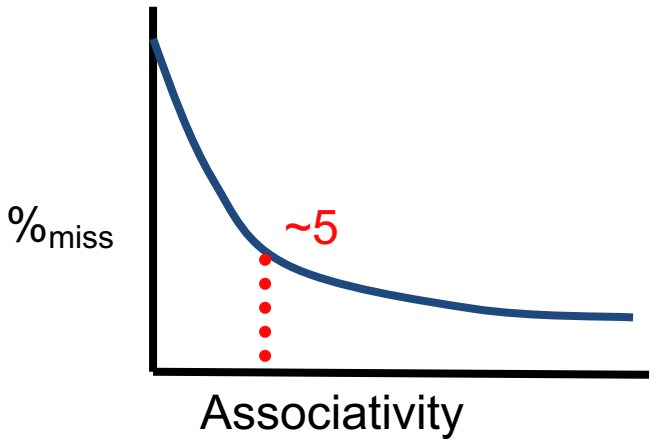
Load: 1110 Miss

Set	Way 0 Tag	Way 0 Data 0	Way 0 Data 1	LRU	Way 1 Tag	Way 1 Data 0	Way 1 Data 1
0	00	A	B	0	01	E	F
1	00	C	D	**0**	**11**	**P**	**Q**

LRU updated on each access
(not just misses)

Associativity and Performance

- Higher associative caches
  - + Have better (lower) $\%_{miss}$
  - - However t_{hit} increase
    - - The more associative, the slower



- Block-size and number of sets should be powers of two
  - – Makes indexing easier

Classifying Misses

- Cold misses (Compulsory misses)
  - First time to access an address
  - Misses even in infinite caches
- Capacity misses
  - Misses due to limited capacity
  - Misses even in fully-associative caches
  - Effect of replacement policy?
- Conflict misses
  - Misses due to limited associativity
- Coherence misses (will explain later in multiprocessors)

Reducing Miss Rates

- Bigger capacity (capacity misses)
- Higher associativity (conflict misses)
- Larger block size (to a certain point)
- Better replacement policies

Handling Cache Writes

- When to propagate new value to (lower level) memory?

- Option #1: Write-through: immediately
  - Update next level of cache (or memory) for every write
  - If cache hit, update cache too
  - May generate many small writes (partial writes 1B-8B) to the same block
  - Can use *coalescing buffers* to reduce write traffics in the next level cache

- Option #2: Write-back: when block is replaced
  - Requires additional "dirty" bit per block
    - Replace clean block: no extra traffic
    - Replace dirty block: extra "writeback" of block

Miss Handling for Writes

- Write-allocate
  - Fill cache for a write miss
  - Expecting more future read/write accessed to the filed block
  - Not useful for isolated writes
  - Used by most of caches

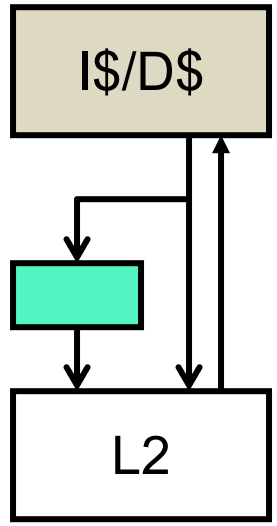
- No Write-allocate
  - Can save bandwidth, but…

Inclusive vs. Exclusive Caches

- Suppose a processor has L1 and L2 caches
- **Inclusive Caches** (completely inclusive)
  - If a block **A** is in L1, then **A** must be in L2
  - On a L1 miss, both caches are filled
  - An eviction or invalidation of a block in L2 → probe L1 and invalidate the block in L1
  - Make sense only when L2 is much bigger than L1
- **Exclusive Caches** (completely exclusive)
  - A block can exist only one-level of cache at a time
  - On a L1 miss, only L1 cache is filled
  - On a L1 eviction, the victim block goes to L2
  - L2 becomes the victim buffer of L1
  - Can save capacity with no duplicate copies across levels of caches
- Less strict inclusivity or exclusivity (mostly inclusive caches?)

Prefetching

- Bring data into cache proactively/**speculatively**
  - If successful, reduces number of cache misses
- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware



- Effectiveness determined by:
  - **Accuracy**: how many prefetches were useful
  - **Coverage**: how many of misses can be reduced
  - **Timeliness**: initiate prefetches sufficiently in advance
- Simple hardware prefetching: **next block prefetching**
  - Miss on address **X** → anticipate miss on **X+block-size**
  + Works for insns: sequential execution
  + Works for data: arrays

Common Prefetchers

- Next line, Adjacent line
- Stride
  - For address `A`, prefetch `A + S x B` (Stride = S x B)
- Pointer
  - Pointer chasing (linked list)
  - Find next address from *data*
- Correlation
  - Find address correlation: A is followed by B
  - Markov chain model
- Region
  - For address `A`, prefetch `A-N`, …, `A-1`, `A+1`, …, `A+N`

Software Prefetching

- Use a special "prefetch" instruction
  - Tells the hardware to bring in data, doesn't actually read it
  - Just a hint
- Inserted by programmer or compiler
- Example

```
int tree_add(tree_t* t) {
  if (t == NULL) return 0;
  __builtin_prefetch(t->left);
  return t->val + tree_add(t->right) + tree_add(t->left);
}
```

- Multiple prefetches bring multiple blocks in parallel
  - More "memory-level" parallelism (MLP)

x86 PREFETCH Instructions

PREFETCH*h*—Prefetch Data Into Caches

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 18 /1	PREFETCHT0 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA *m8*	Valid	Valid	Move data from *m8* closer to the processor using NTA hint.

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - Pentium III processor—1st- or 2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - Pentium III processor—1st-level cache
  - Pentium 4 and Intel Xeon processors—2nd-level cache

microarchitecture dependent specification

different instructions for different cache levels

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
  - Several code restructuring techniques to improve both
    - Loop blocking (break into cache-sized chunks), loop fusion
  - Compiler must know that restructuring preserves semantics
- **Loop interchange**: spatial locality
  - Example: for a row-major matrix, `X[i][j]` should be 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 (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j++)
      sum += X[i][j];
```

Software Restructuring: Data

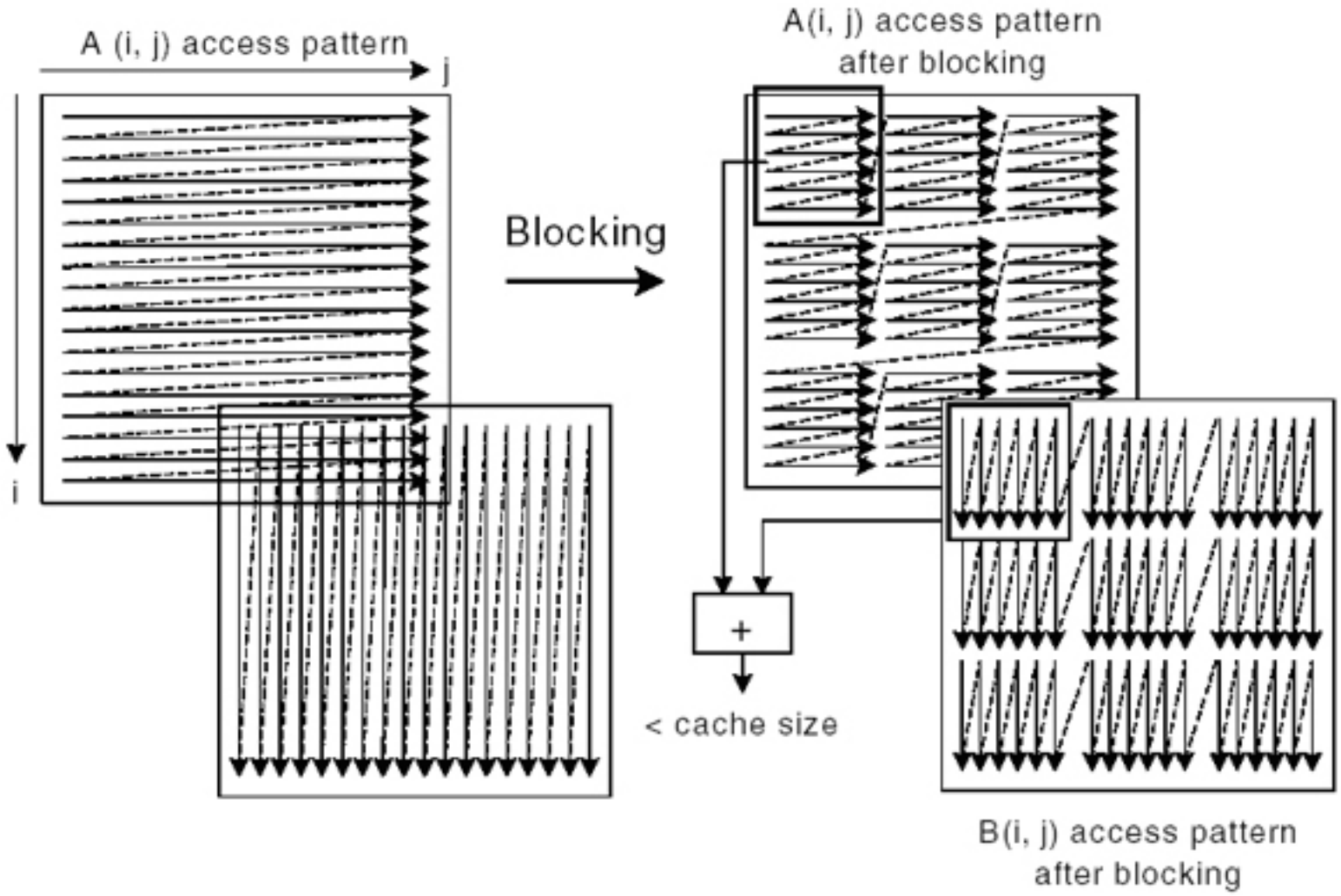
- **Loop blocking**: temporal locality
  - Poor code

```
for (k=0; k<NUM_ITERATIONS; k++)
   for (i=0; i<NUM_ELEMS; i++)
      X[i] = f(X[i]);    // say
```

  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

```
for (i=0; i<NUM_ELEMS; i+=CACHE_SIZE)
   for (k=0; k<NUM_ITERATIONS; k++)
      for (j=0; j<CACHE_SIZE; j++)
         X[i+j] = f(X[i+j]);
```

  - Assumes you know **CACHE_SIZE**, do you?



Credit: How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel® Architecture

Summary

- Basic cache trade-offs and performance metric
- In recent microprocessors, caches use more than 50% of chip area → memory system performance is critical
- No magic solution yet for hit latency vs. miss rate
- Prefetcher can hide the future cache misses

- Next lecture: Supporting virtual memory