

ARCHITECTURE OF  
COMPUTER SYSTEMS  
LECTURE 7 - MEMORY

HIERARCHY-II

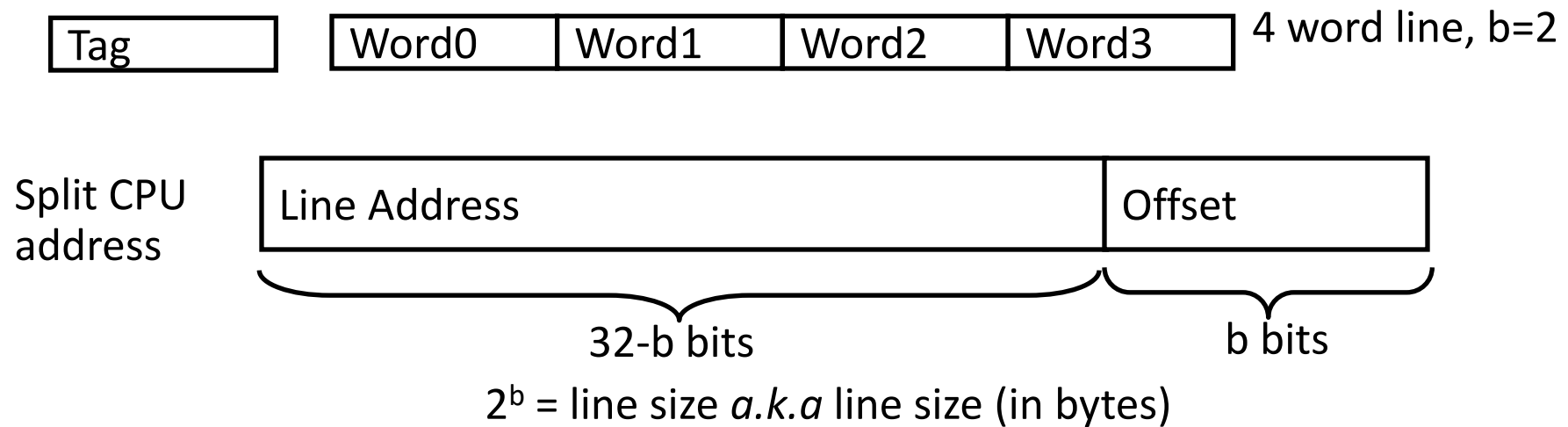


# LAST TIME IN LECTURE 6

- Dynamic RAM (DRAM) is main form of main memory storage in use today
  - Holds values on small capacitors, need refreshing (hence dynamic)
  - Slow multi-step access: precharge, read row, read column
- Static RAM (SRAM) is faster but more expensive
  - Used to build on-chip memory for caches
- Cache holds small set of values in fast memory (SRAM) close to processor
  - Need to develop search scheme to find values in cache, and replacement policy to make space for newly accessed locations
- Caches exploit two forms of predictability in memory reference streams
  - Temporal locality, same location likely to be accessed again soon
  - Spatial locality, neighboring location likely to be accessed soon

# LINE SIZE AND SPATIAL LOCALITY

A line is unit of transfer between the cache and memory



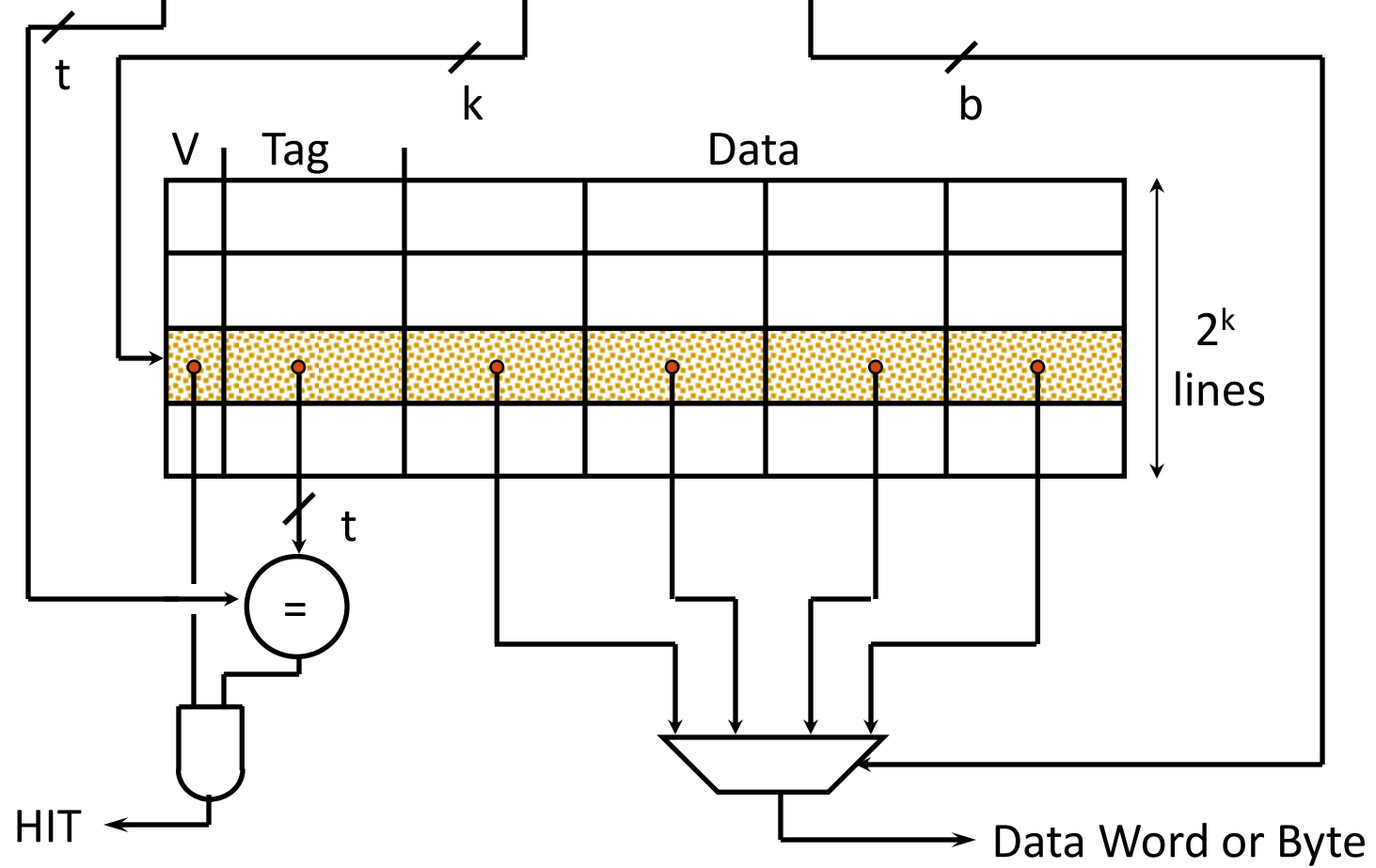
Larger line size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

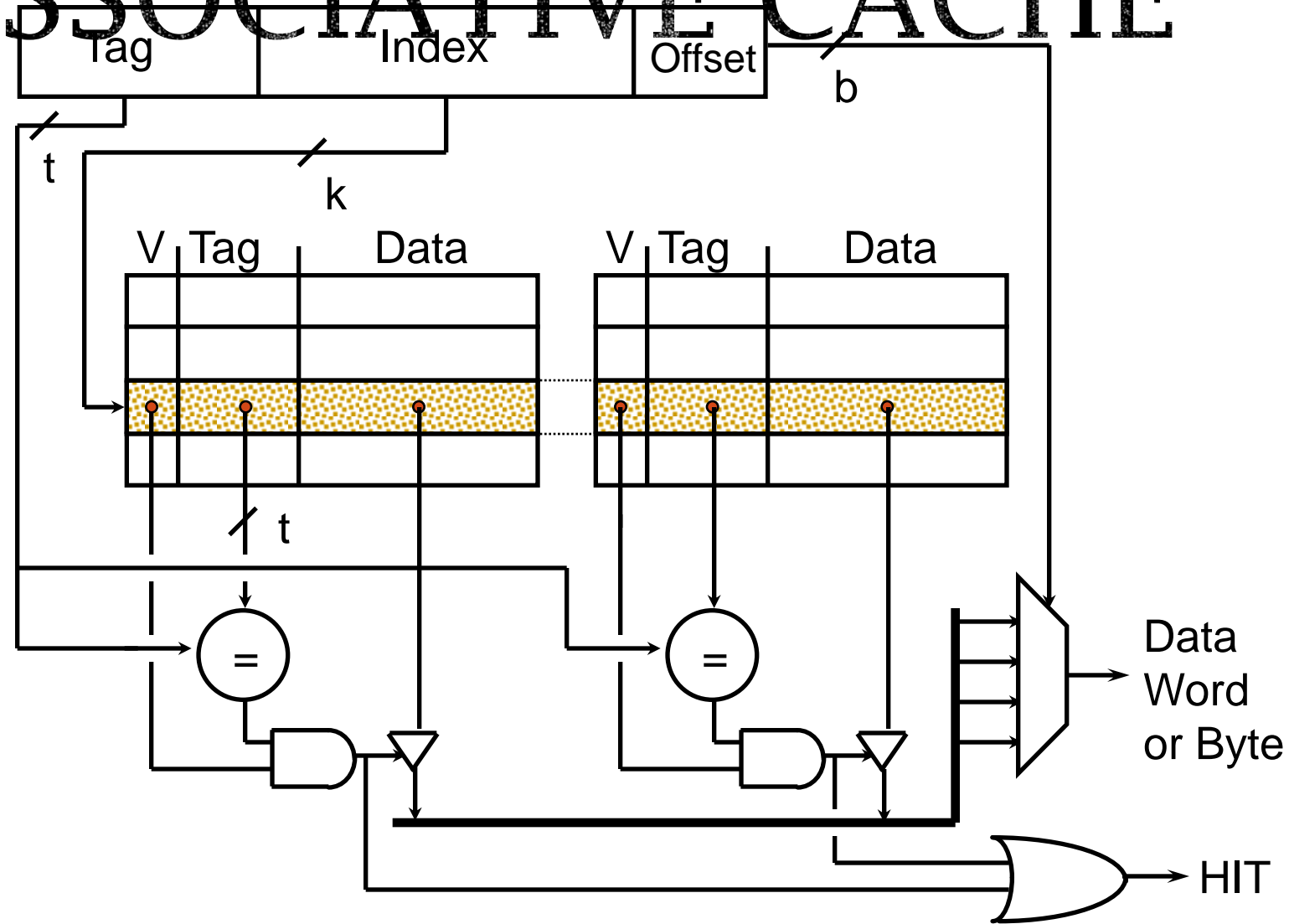
*What are the disadvantages of increasing line size?*

*Fewer lines => more conflicts. Can waste bandwidth.*

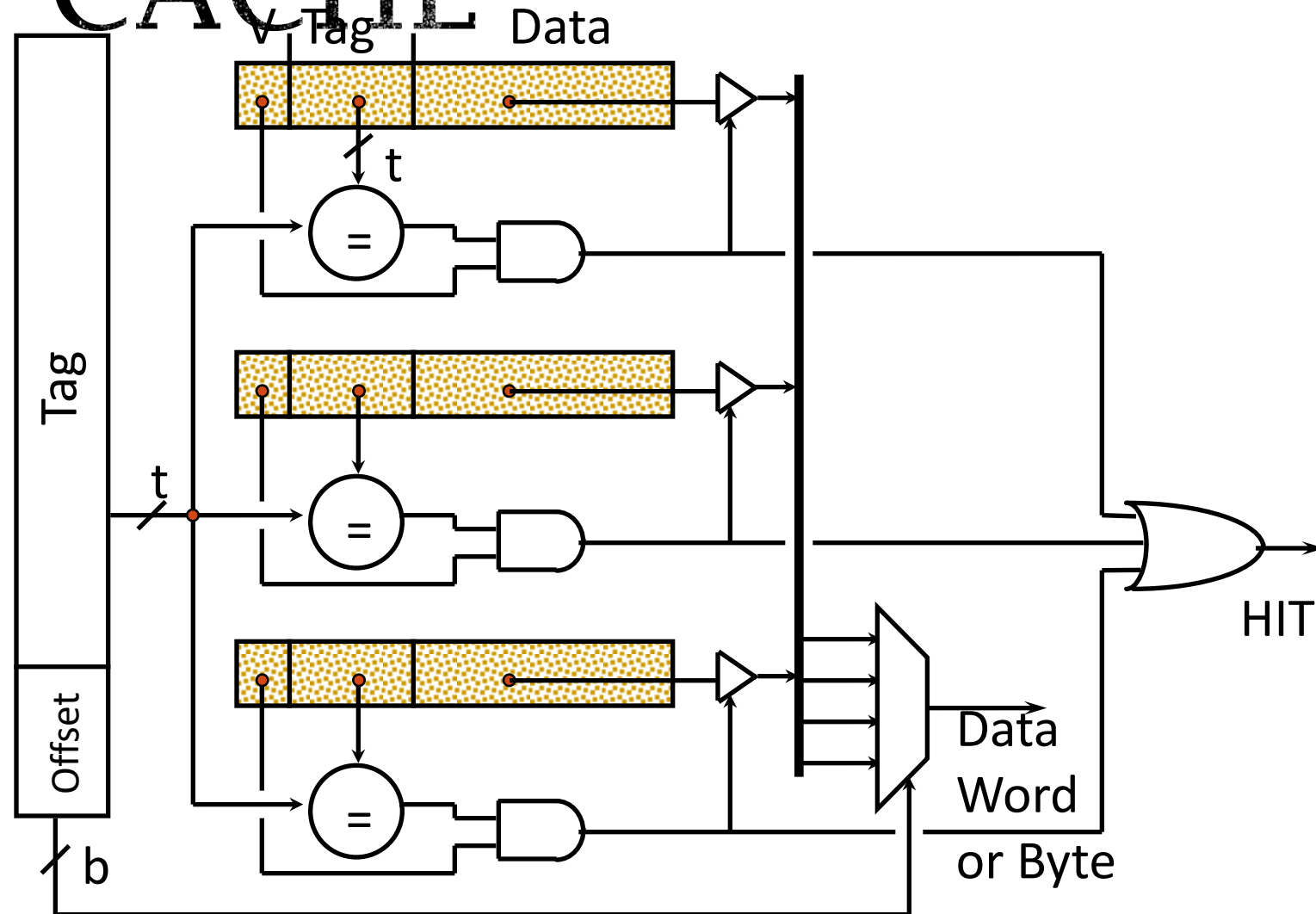
# DIRECT-MAPPED CACHE



# 2-WAY SET-ASSOCIATIVE CACHE



# FULLY ASSOCIATIVE CACHE



# REPLACEMENT POLICY

In an associative cache, which line from a set should be evicted when the set becomes full?

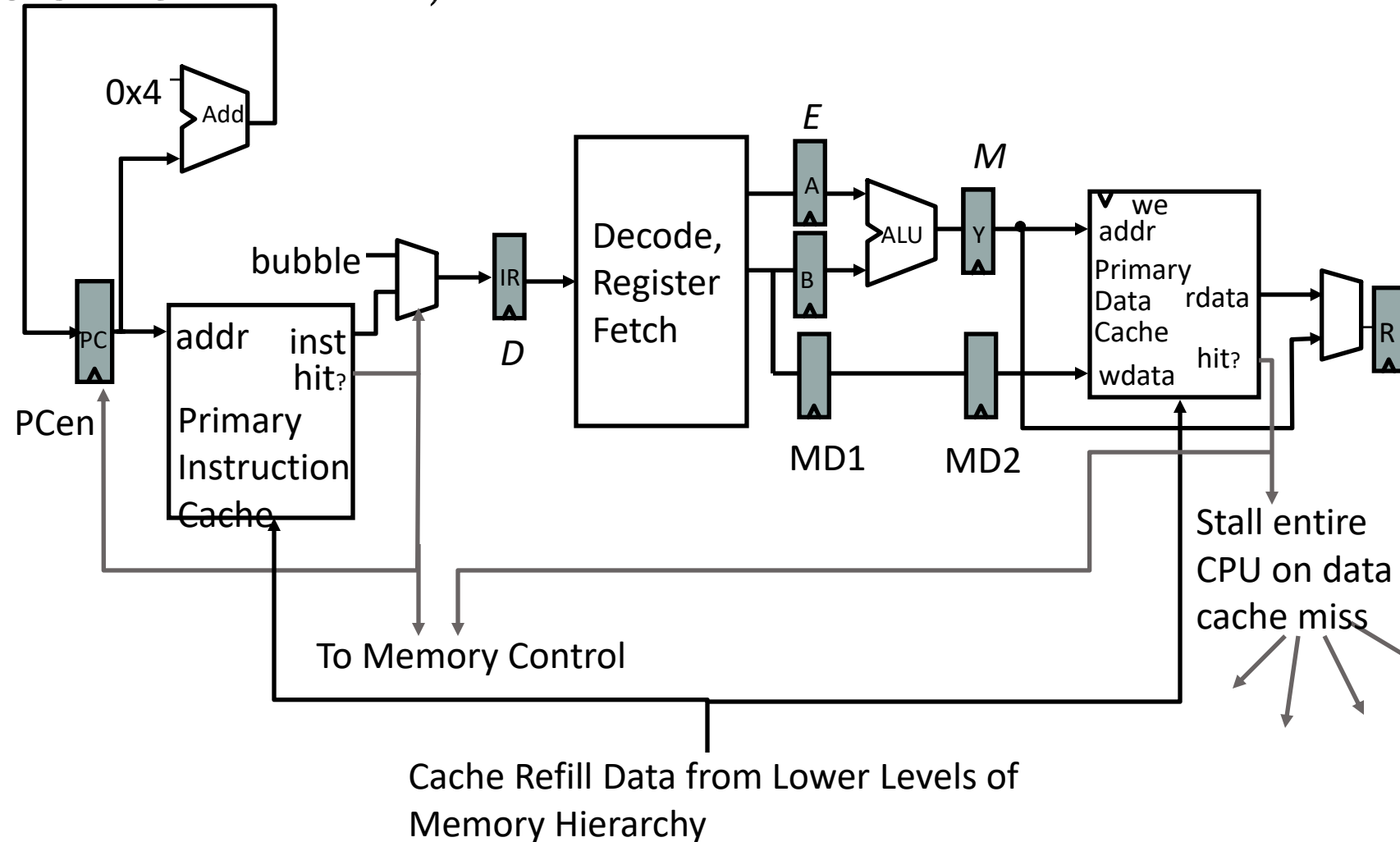
- Random
- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - True implementation only feasible for small sets (2-way)
  - Pseudo-LRU binary tree often used for 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - Used in highly associative caches
- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used line or lines

*This is a second-order effect. Why?*

*Replacement only happens on misses*

# CPU-CACHE INTERACTION

(5-STAGE PIPELINE)





# IMPROVING CACHE

Average memory access time (AMAT) =  
 $\text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$

# PERFORMANCE

To improve performance:

- reduce the hit time
- reduce the miss rate
- reduce the miss penalty

*What is best cache design for 5-stage pipeline?*

*Biggest cache that doesn't increase hit time past 1 cycle  
(approx 8-32KB in modern technology)*

*[ design issues more complex with deeper pipelines and/or out-of-order superscalar processors ]*

# CAUSES OF CACHE

## MISSES: THE 3 C'S

**Compulsory:** first reference to a line (a.k.a. cold start misses)

- *misses that would occur even with infinite cache*

**Capacity:** cache is too small to hold all data needed by the program

- *misses that would occur even under perfect replacement policy*

**Conflict:** misses that occur because of collisions due to line-placement strategy

- *misses that would not occur with ideal full associativity*

# EFFECT OF CACHE PARAMETERS ON PERFORMANCE

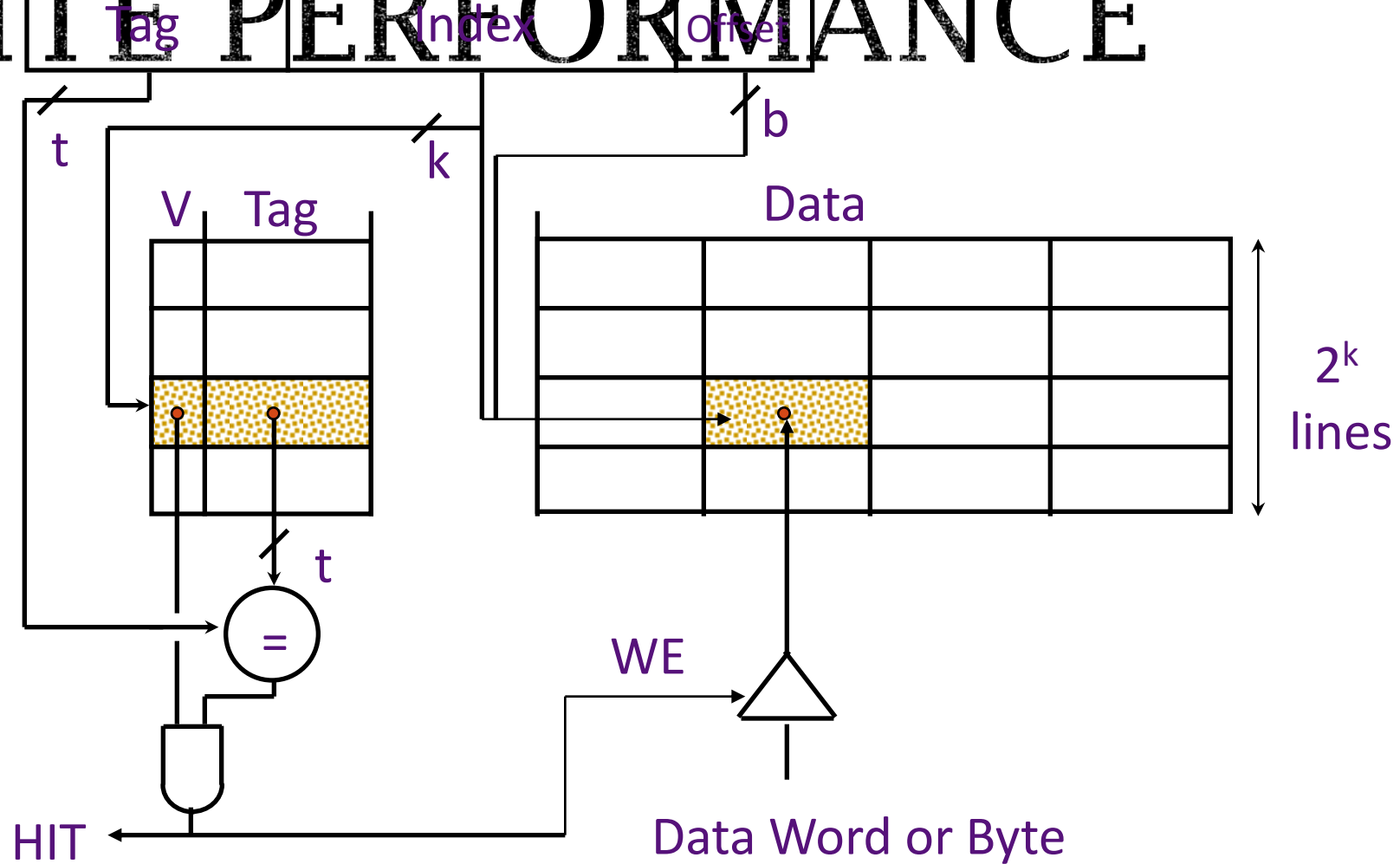
- Larger cache size
  - + reduces capacity and conflict misses
  - hit time will increase

- Higher associativity
  - + reduces conflict misses
  - may increase hit time
- Larger line size
  - + reduces compulsory and capacity (reload) misses
  - increases conflict misses and miss penalty

# WRITE POLICY CHOICES

- **Cache hit:**
  - ***write through***: write both cache & memory
    - Generally higher traffic but simpler pipeline & cache design
  - ***write back***: write cache only, memory is written only when the entry is evicted
    - A dirty bit per line further reduces write-back traffic
    - Must handle 0, 1, or 2 accesses to memory for each load/store
- **Cache miss:**
  - ***no write allocate***: only write to main memory
  - ***write allocate*** (aka fetch on write): fetch into cache
- **Common combinations:**
  - write through and no write allocate
  - write back with write allocate

# WRITE PERFORMANCE



# REDUCING WRITE HIT TIME

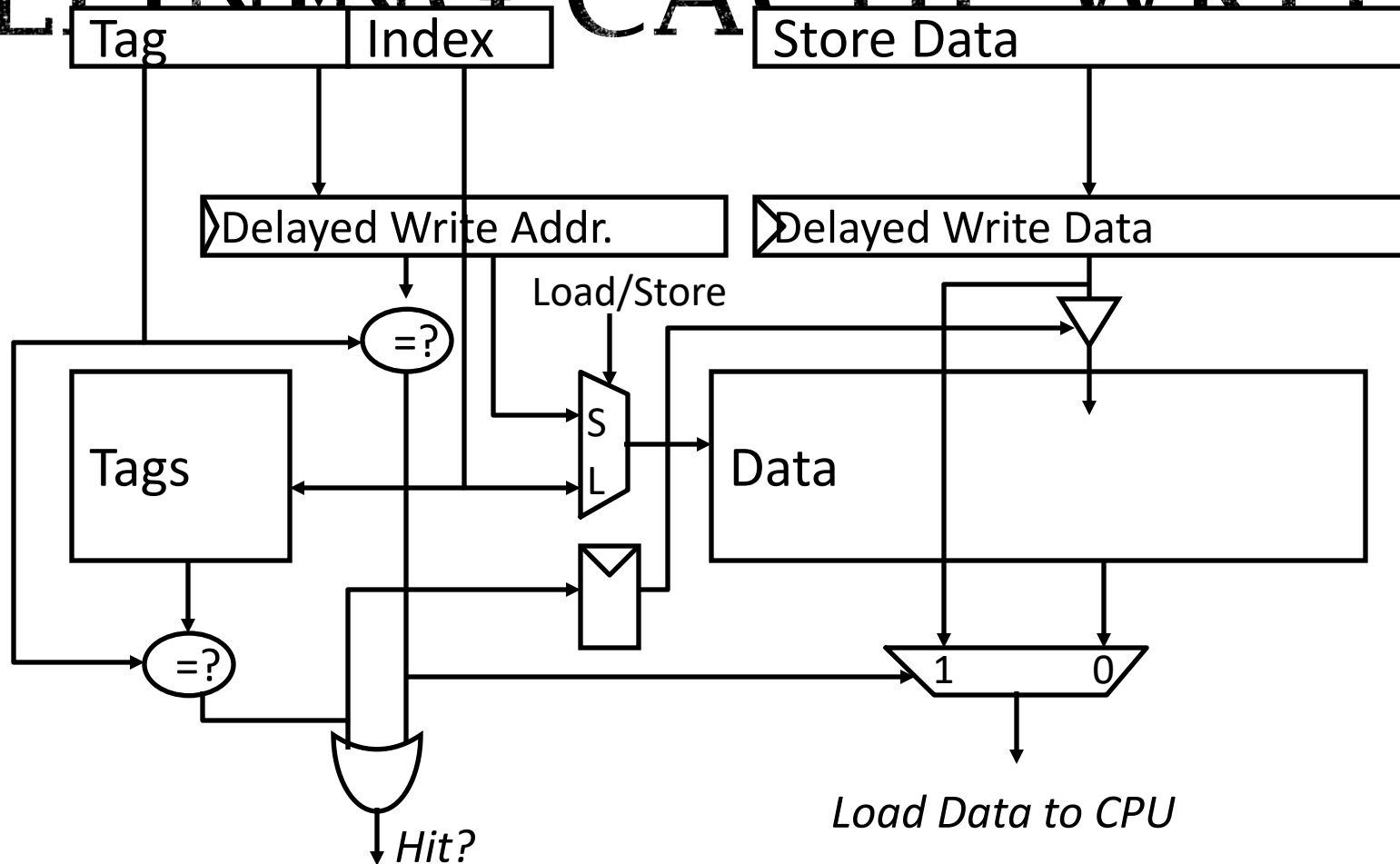
**Problem:** Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

## **Solutions:**

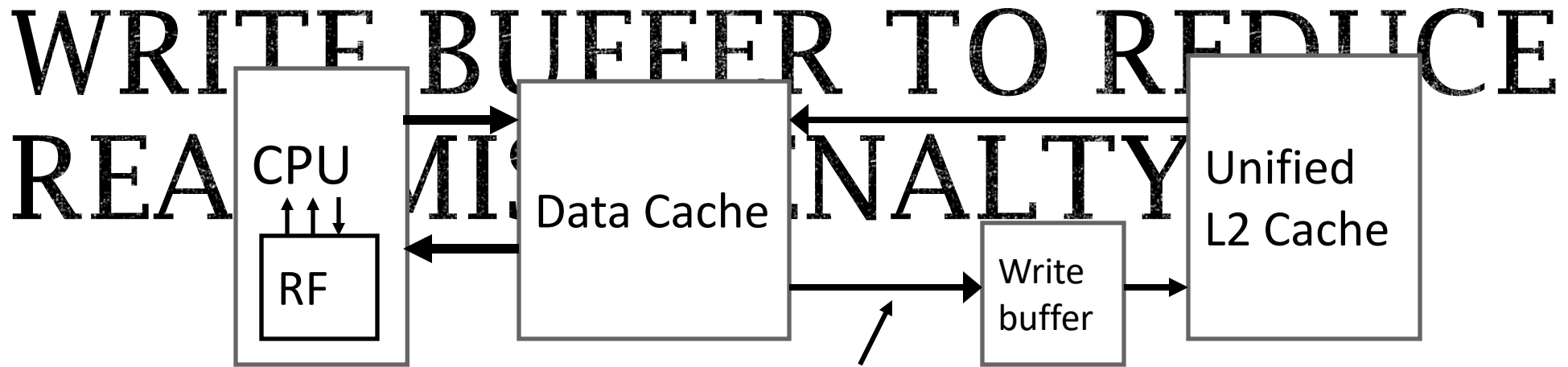
- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- Fully-associative (CAM Tag) caches: Word line only enabled if hit
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check

# PIPELINING CACHE WRITES

*Address and Store Data From CPU*



*Data from a store hit written into data portion of cache during tag access of subsequent store*



Evicted dirty lines for writeback cache

OR

All writes in writethrough cache

Processor is not stalled on writes, and read misses can go ahead of write to main memory

**Problem:** Write buffer may hold updated value of location needed by a read miss

**Simple solution:** on a read miss, wait for the write buffer to go empty

**Faster solution:** Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer



# OVERHEAD WITH SUB-BLOCKS

- **Problem:** Tags are too large, i.e., too much overhead
  - Simple solution: Larger lines, but miss penalty could be large.
- **Solution:** Sub-block placement (aka sector cache)
  - A valid bit added to units smaller than full line, called sub-blocks
  - Only read a sub-block on a miss
  - *If a tag matches, is the word in the cache?*

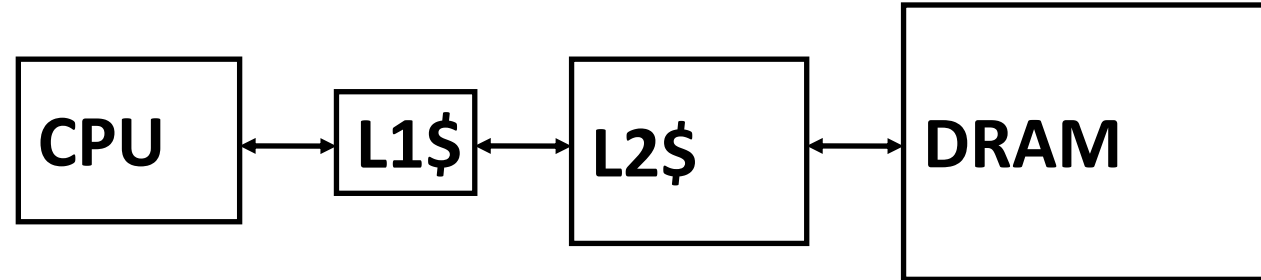
<i>100</i>
<i>300</i>
<i>204</i>

<i>1</i>		<i>1</i>		<i>1</i>		<i>1</i>	
<i>1</i>		<i>1</i>		<i>0</i>		<i>0</i>	
<i>0</i>		<i>1</i>		<i>0</i>		<i>1</i>	

# MULTILEVEL CACHES

Problem: A memory cannot be large and fast

Solution: Increasing sizes of cache at each level



Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions

# PRESENCE OF L2 INFLUENCES L1 DESIGN

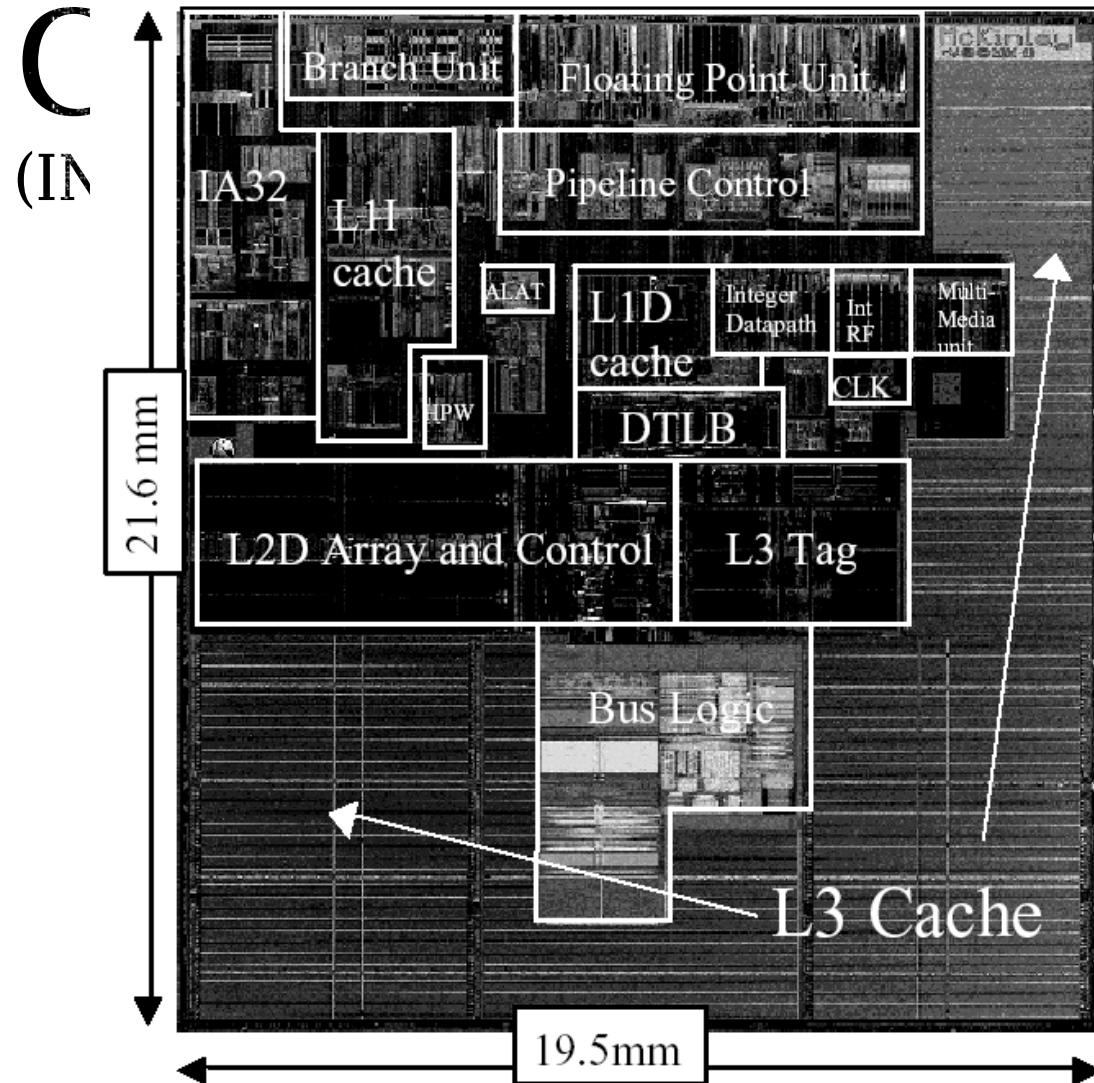
- Use smaller L1 if there is also L2
  - Trade increased L1 miss rate for reduced L1 hit time
  - Backup L2 reduces L1 miss penalty
  - Reduces average access energy
- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cache absorbs write traffic, doesn't go off-chip
  - At most one L1 miss request per L1 access (no dirty victim write back) simplifies pipeline control
  - Simplifies coherence issues
  - Simplifies error recovery in L1 (can use just parity bits in L1 and reload from L2 when parity error detected on L1 read)

# INCLUSIVE MULTILEVEL CACHE:

- Inclusive multilevel cache:
  - Inner cache can only hold lines also present in outer cache
  - External coherence snoop access need only check outer cache
- *Exclusive* multilevel caches:
  - Inner cache may hold lines not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

Why choose one type or the other?

# ITANIUM-2 ON-CHIP



**Level 1:** 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

**Level 2:** 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency

**Level 3:** 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

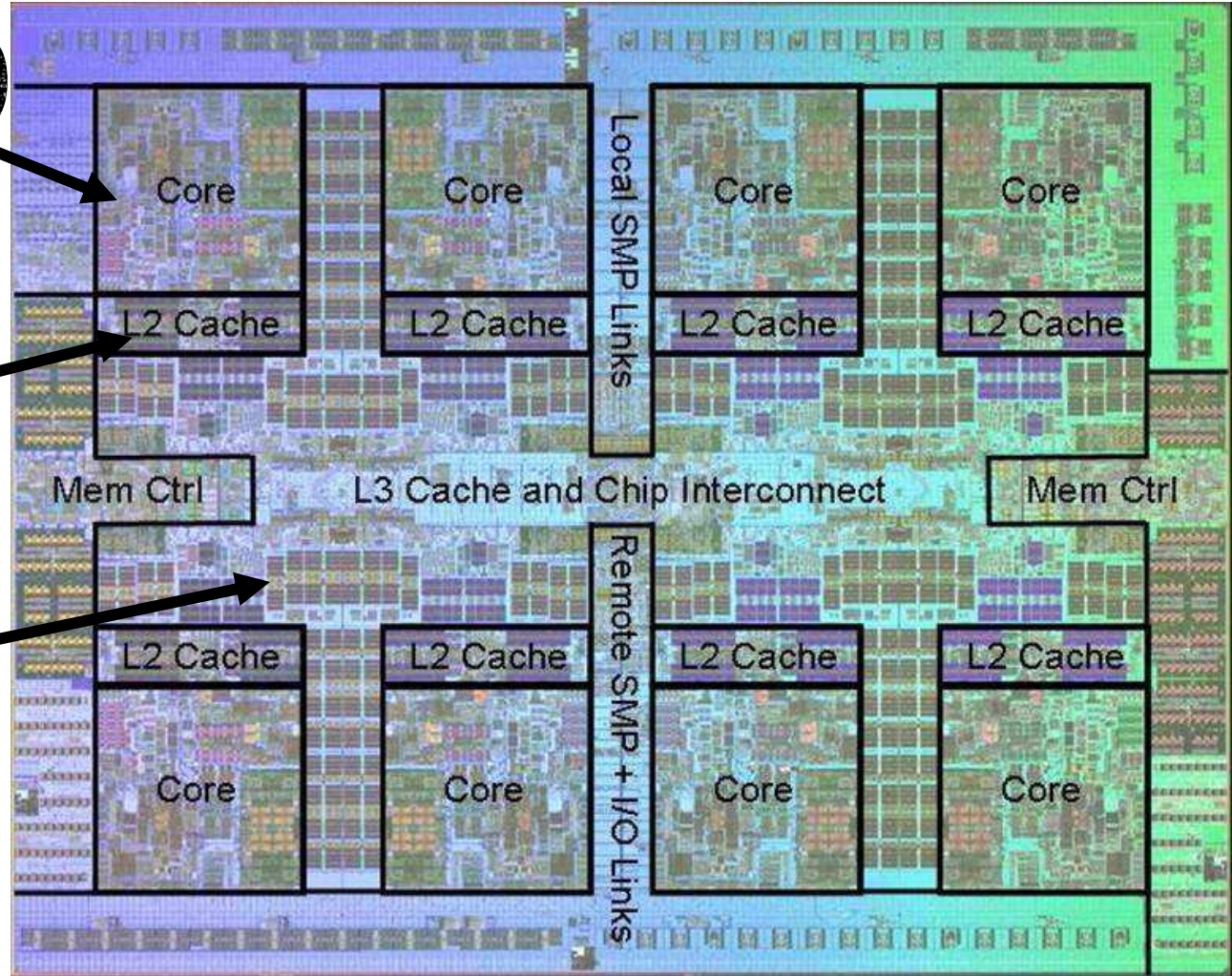


# POWER 7 ON-CHIP CACHES

32KB L1 I\$/core  
32KB L1 D\$/core  
3-cycle latency  
**[IBM 2009]**

256KB Unified L2\$/core  
8-cycle latency

32MB Unified Shared L3\$  
Embedded DRAM (eDRAM)  
25-cycle latency to local slice



# IBM Z196 MAINFRAME

## CACHES 2010

- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-\$/core + 128KB D-\$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)

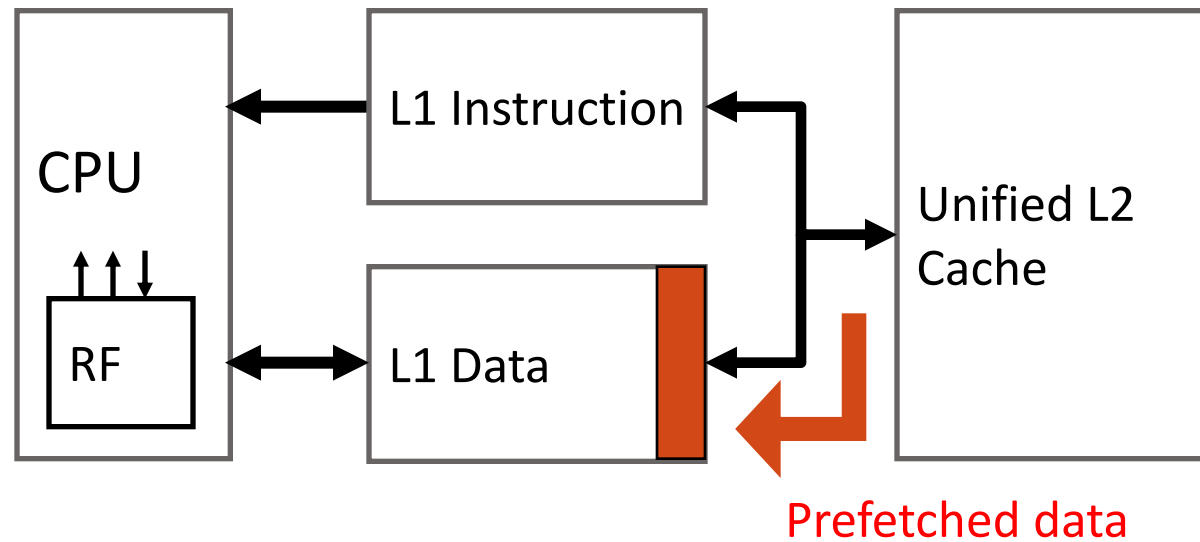
# PREFETCHING

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- What types of misses does prefetching affect?



# ISSUES IN PREFETCHING

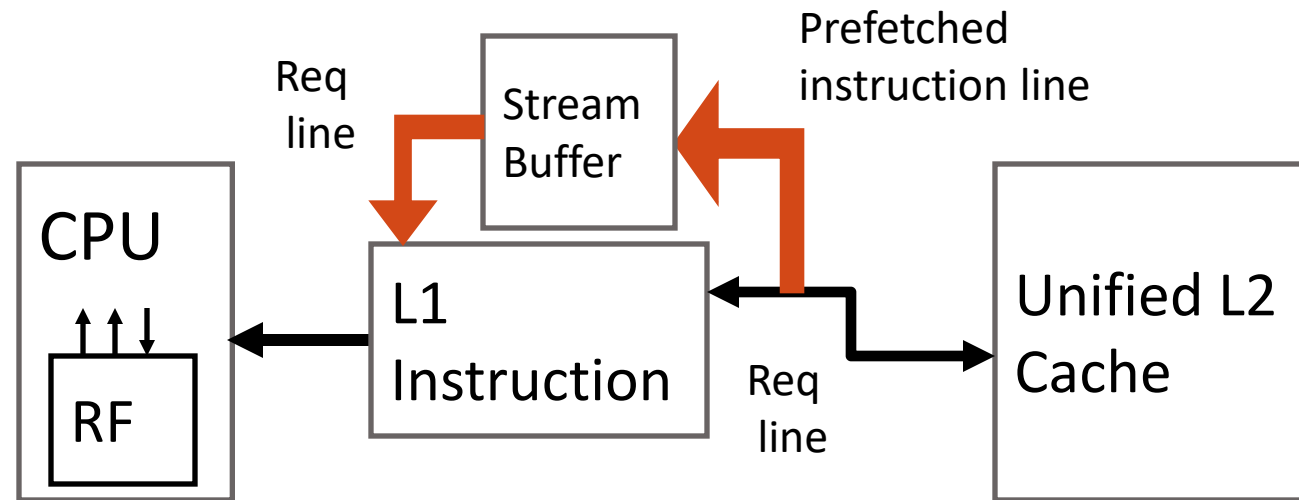
- Usefulness - should produce hits
- Timeliness - not late and not too early
- Cache and bandwidth pollution



# HARDWARE INSTRUCTION

## Instruction prefetch in Alpha AXP 21064

- Fetch two lines on a miss; the requested line (i) and the next consecutive line (i+1)
- Requested line placed in cache, and next line in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer line into cache and prefetch next line (i+2)



# HARDWARE DATA PREFETCHING

- Prefetch-on-miss:
  - Prefetch  $b + 1$  upon miss on  $b$
- One-Block Lookahead (OBL) scheme
  - Initiate prefetch for block  $b + 1$  when block  $b$  is accessed
  - Why is this different from doubling block size?
  - Can extend to  $N$ -block lookahead
- Strided prefetch
  - If observe sequence of accesses to line  $b$ ,  $b+N$ ,  $b+2N$ , then prefetch  $b+3N$  etc.
- Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

# SOFTWARE PREFETCHING

```
for (i=0; i < N; i++) {  
    prefetch( &a[i + 1] );  
    prefetch( &b[i + 1] );  
    SUM = SUM + a[i] * b[i];  
}
```

# SOFTWARE PREFETCHING

## ISSUES

- Timing is the biggest issue, not predictability
- If you prefetch very close to when the data is required, you might be too late
- Prefetch too early, cause pollution
- Estimate how long it will take for the data to come into L1, so we can set P appropriately
- *Why is this hard to do?*

```
for(i=0; i < N; i++) {  
    prefetch( &a[i + P] );  
    prefetch( &b[i + P] );  
    SUM = SUM + a[i] * b[i];  
}
```

***Must consider cost of prefetch instructions***

# COMPILER OPTIMIZATIONS

- Restructuring code affects the data access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality
- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before being replaced
  - Needs mechanism for software to tell hardware not to cache data (“no-allocate” instruction hints or page table bits)
- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead cache locations

# LOOP INTERCHANGE

```
for (j=0; j < N; j++) {  
    for (i=0; i < M; i++) {  
        x[i][j] = 2 * x[i][j];  
    }  
}
```



```
for (i=0; i < M; i++) {  
    for (j=0; j < N; j++) {  
        x[i][j] = 2 * x[i][j];  
    }  
}
```

*What type of locality does this improve?*

# LOOP FUSION

```
for(i=0; i < N; i++)  
    a[i] = b[i] * c[i];
```

```
for(i=0; i < N; i++)  
    d[i] = a[i] * c[i];
```



```
for(i=0; i < N; i++)  
{  
    a[i] = b[i] * c[i];  
    d[i] = a[i] * c[i];  
}
```

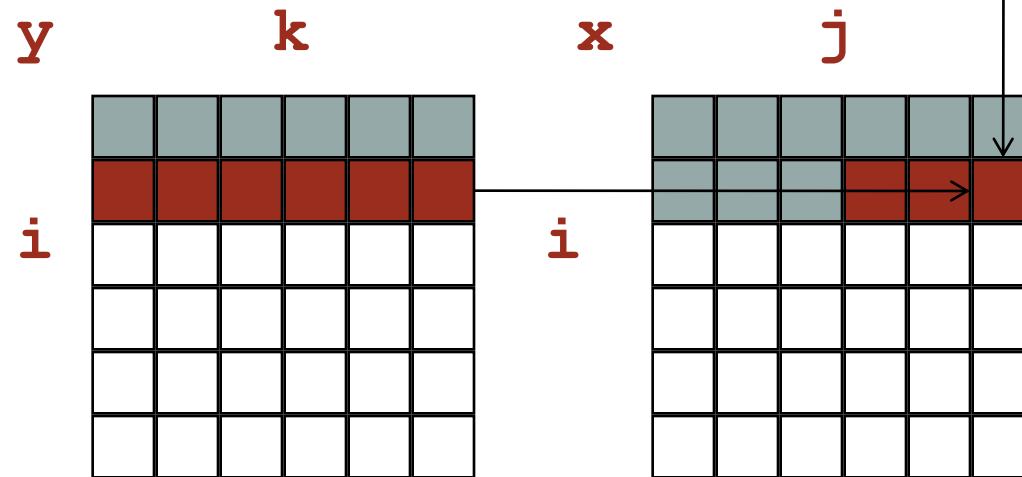
*What type of locality does this improve?*



# MATRIX MULTIPLY, NAÏVE

CODE

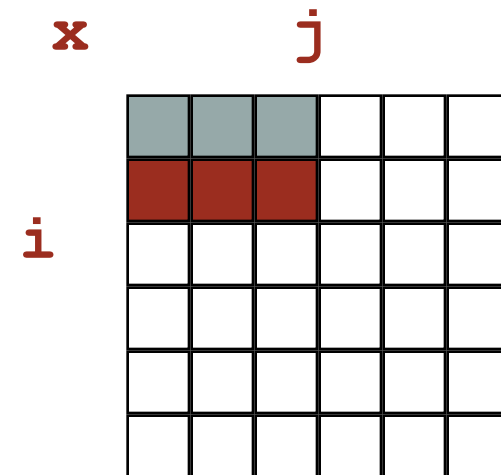
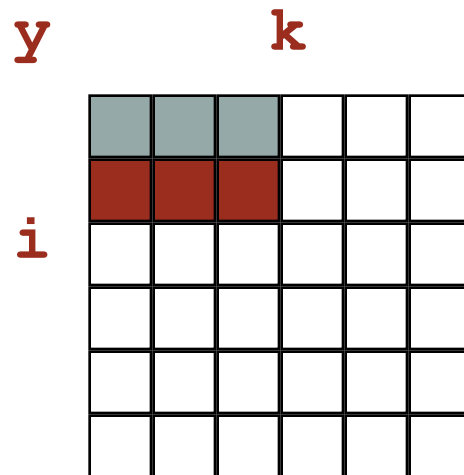
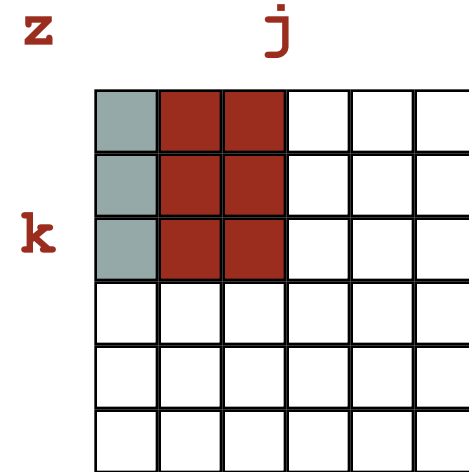
```
for (i=0; i < N; i++)  
  for (j=0; j < N; j++) {  
    r = 0;  
    for (k=0; k < N; k++)  
      r = r + y[i][k] * z[k][j];  
    x[i][j] = r;  
  }
```



□ *Not touched*    ■ *Old access*    ■ *New access*

# MATRIX MULTIPLY WITH CACHE TILING

```
for (jj=0; jj < N; jj=jj+B)
  for (kk=0; kk < N; kk=kk+B)
    for (i=0; i < N; i++)
      for (j=jj; j < min(jj+B,N); j++) {
        r = 0;
        for (k=kk; k < min(kk+B,N); k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = x[i][j] + r;
      }
```



*What type of locality does this improve?*

# ACKNOWLEDGEMENTS

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252