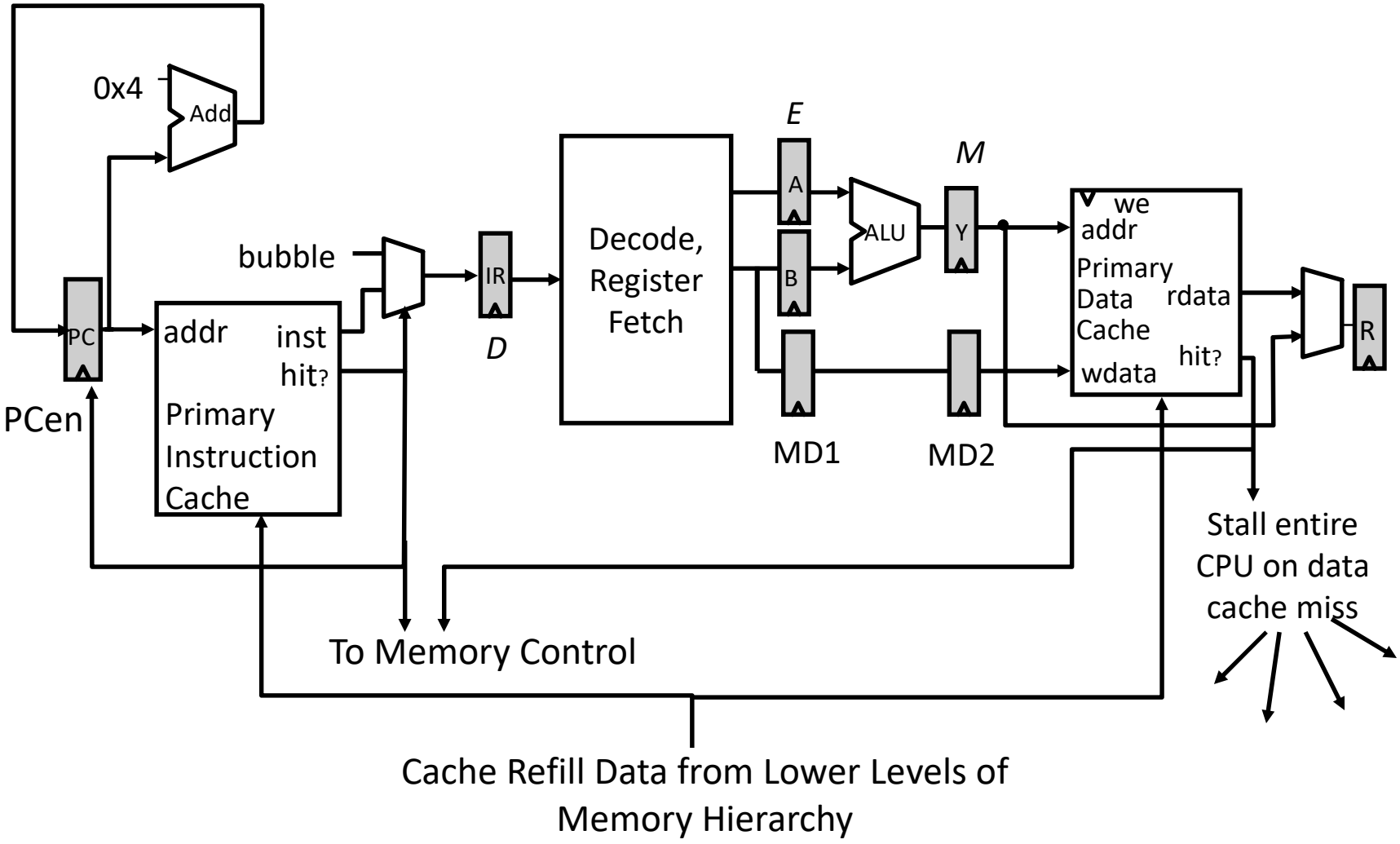
Computer Architecture

Lecture 06 – Advanced Cache

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Recap：CPU-Cache Interaction
(5-stage pipeline)



Agenda

Advanced Cache Optimizations

① Pipelined Cache Write

② Write Buffer

③ Multilevel Caches

④ Victim/Filter/Stream Caches

⑤ Prefetching(hardware/software)

⑥ Multiporting and Banking

⑦ Software Optimizations

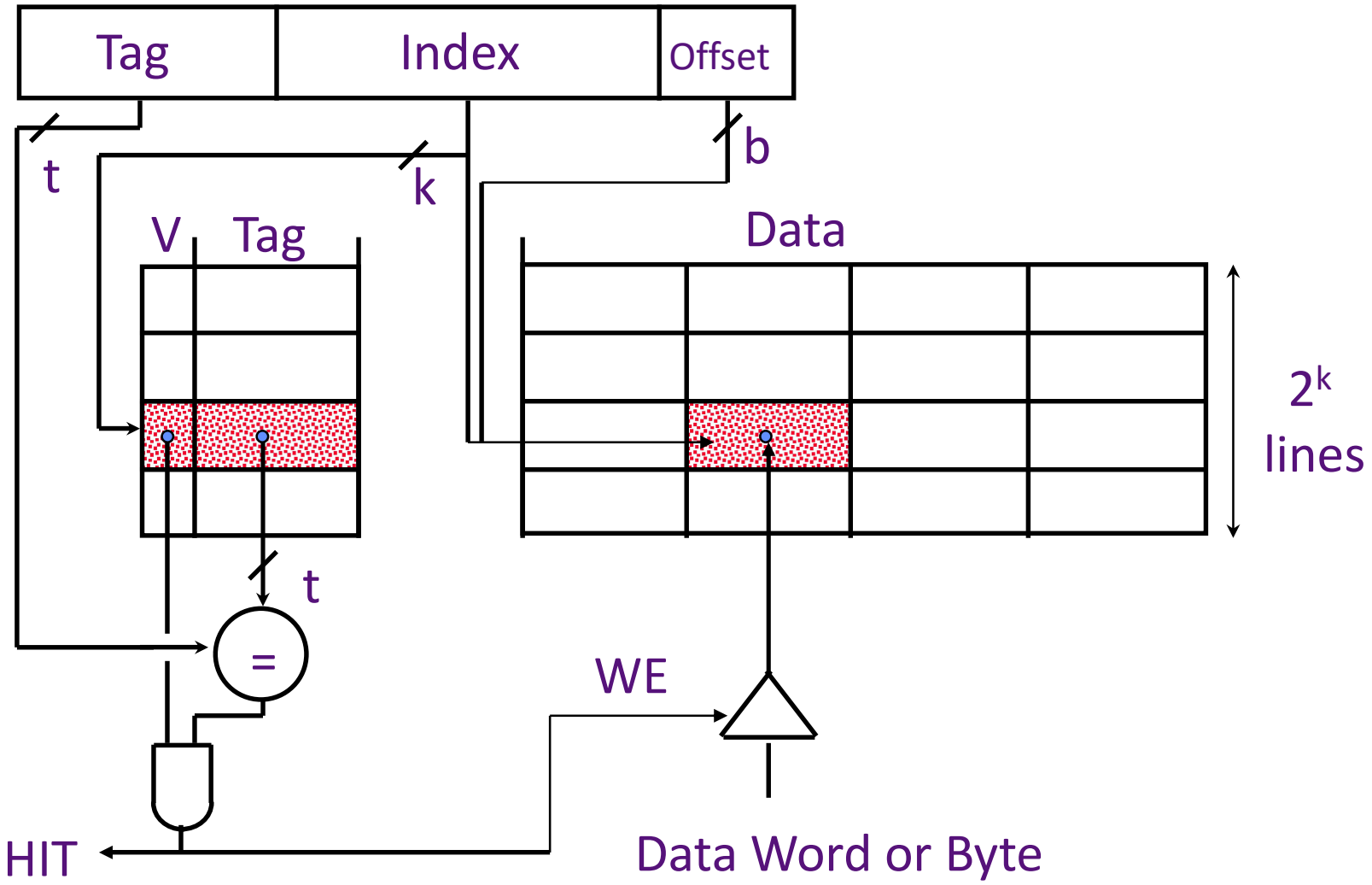
⑧ Non-Blocking Cache

⑨ Critical Word First/Early Restart

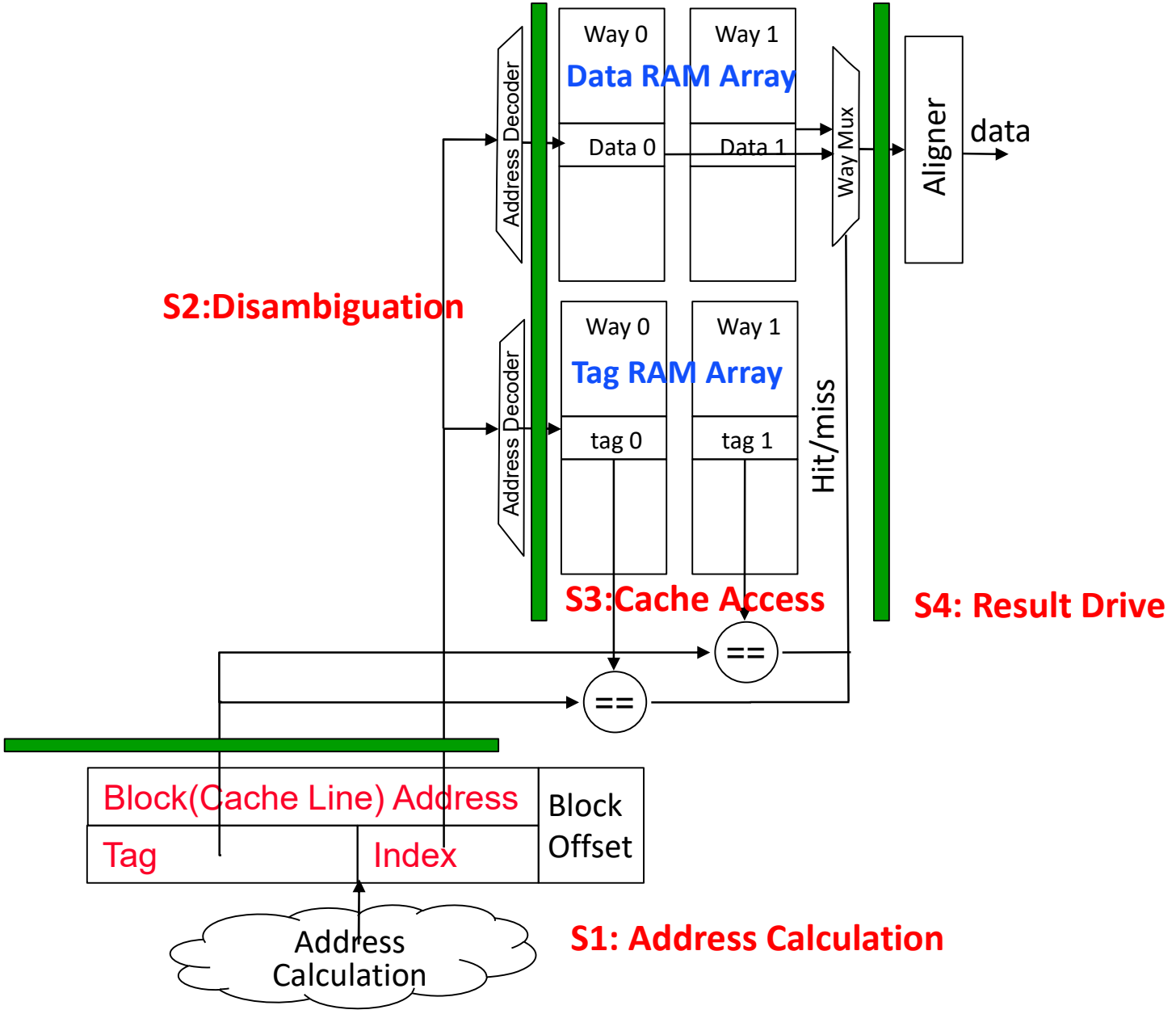
Five Categories of Cache techniques

Effect	Techniques
Reducing the hit time	Small and simple L1 Cache
	Way Predication
Increasing Cache Bandwidth	Pipelined Cache
	Multibanked Caches
	Non-blocking Cache
Reducing the miss penalty	Critical Word first
	Merging write Buffers
Reducing the miss rate	Compiler/Program Optimization
Reducing the miss penalty or miss rate via Parallelism	Hardware Prefetching
	Compiler Prefetching

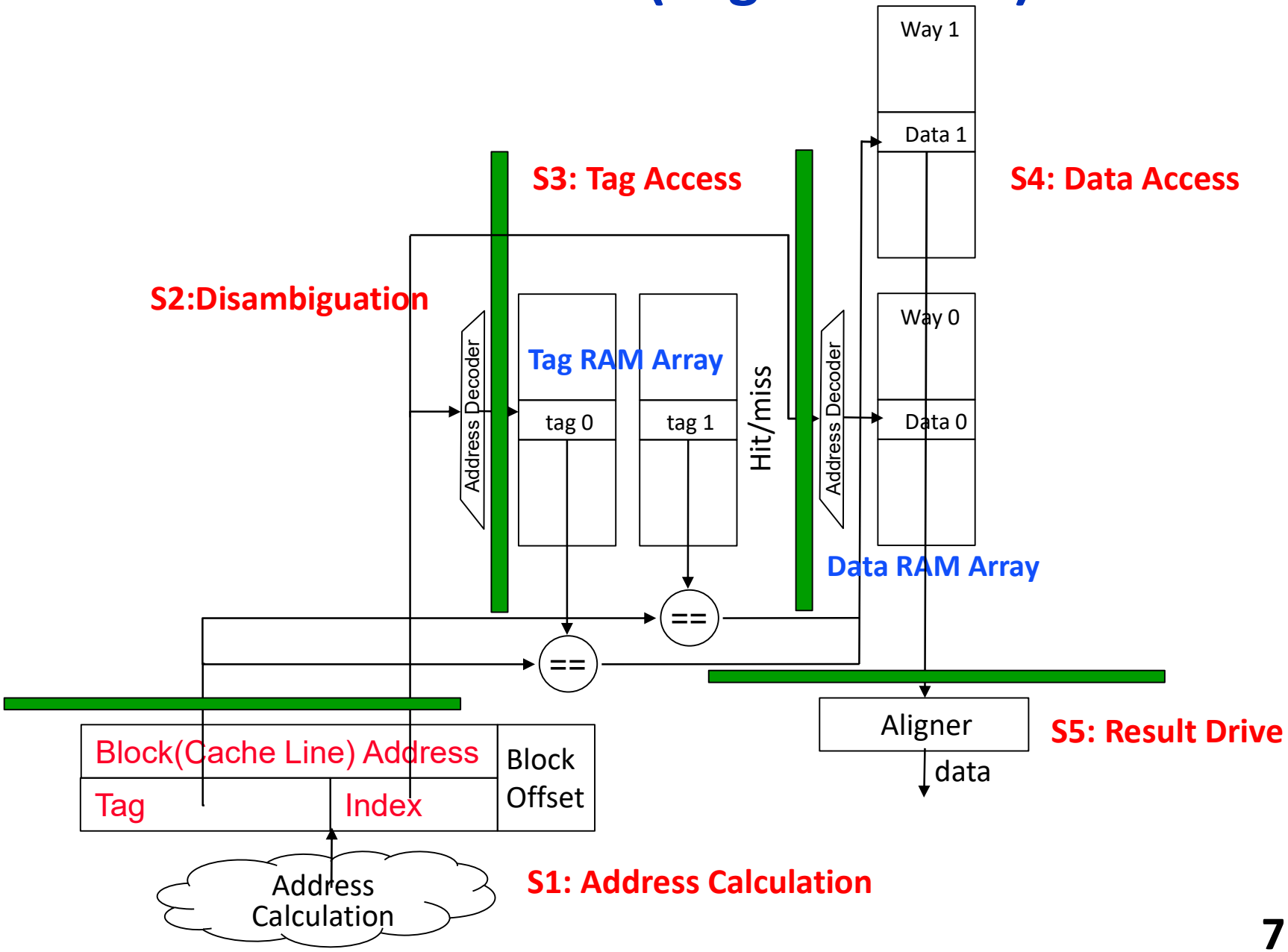
① Pipelined Cache Write : Write Performance



Parallel Cache access (Tag and Data)



Serial Cache access (Tag and Data)



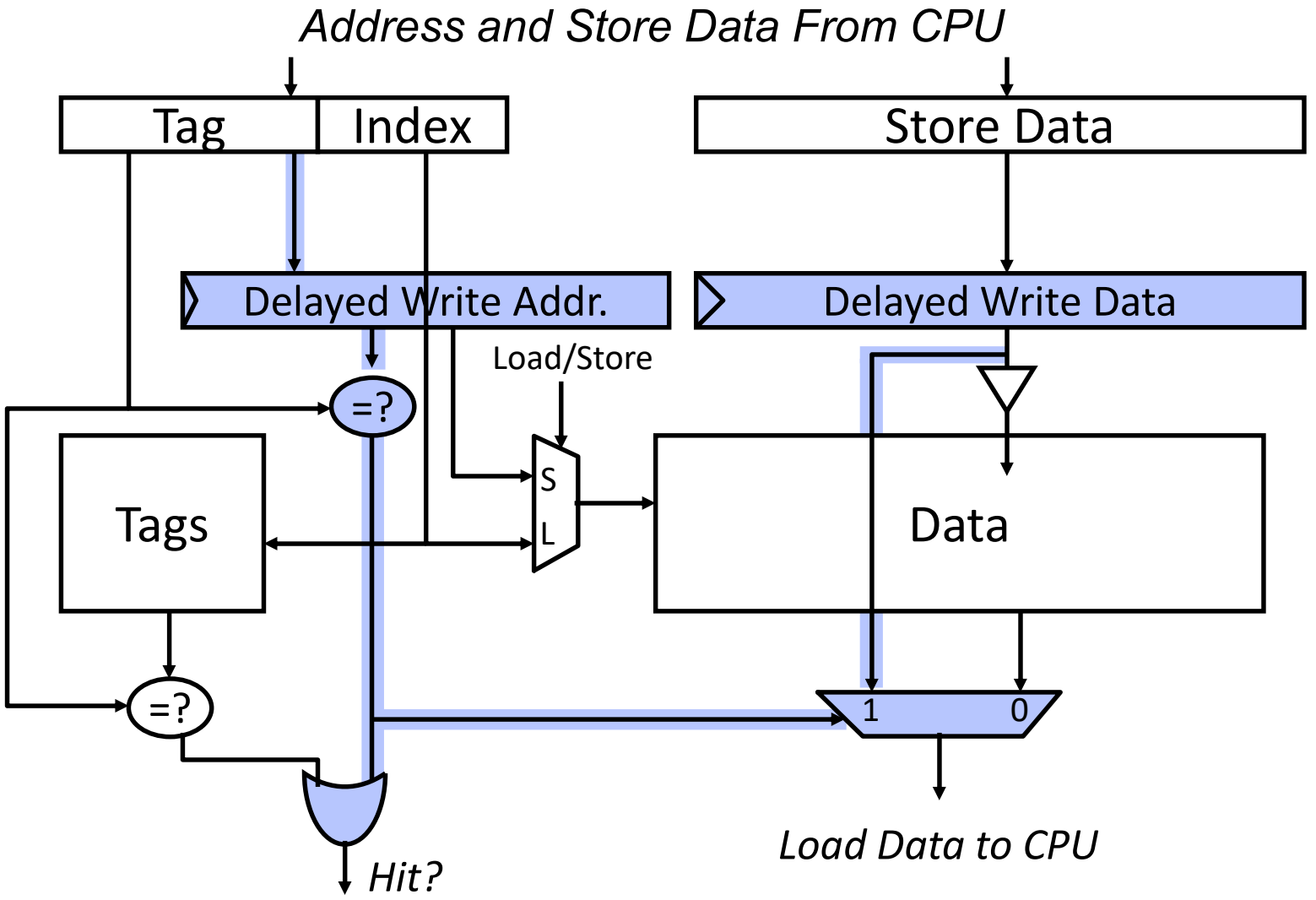
①Pipelined Cache Write：Reducing Write Hit Time

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

Solutions:

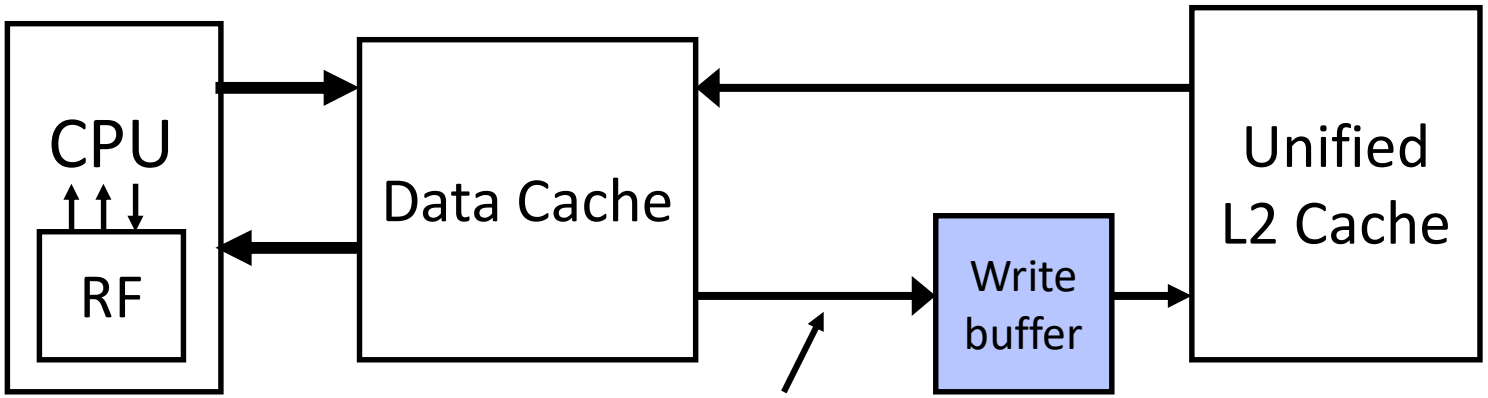
- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- Fully-associative (CAM Tag) caches: Word line only enabled if hit
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check

① Pipelining Cache Writes



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

②Write Buffer to Reduce Read Miss Penalty



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

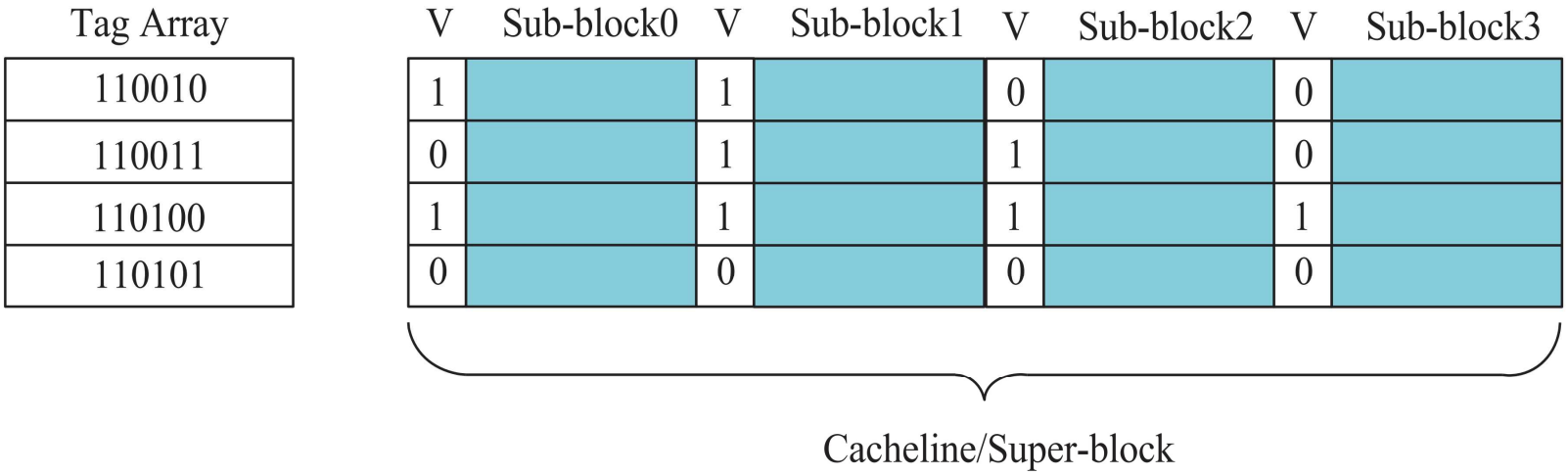
Problem: Write buffer may hold updated value of location needed by a read miss

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

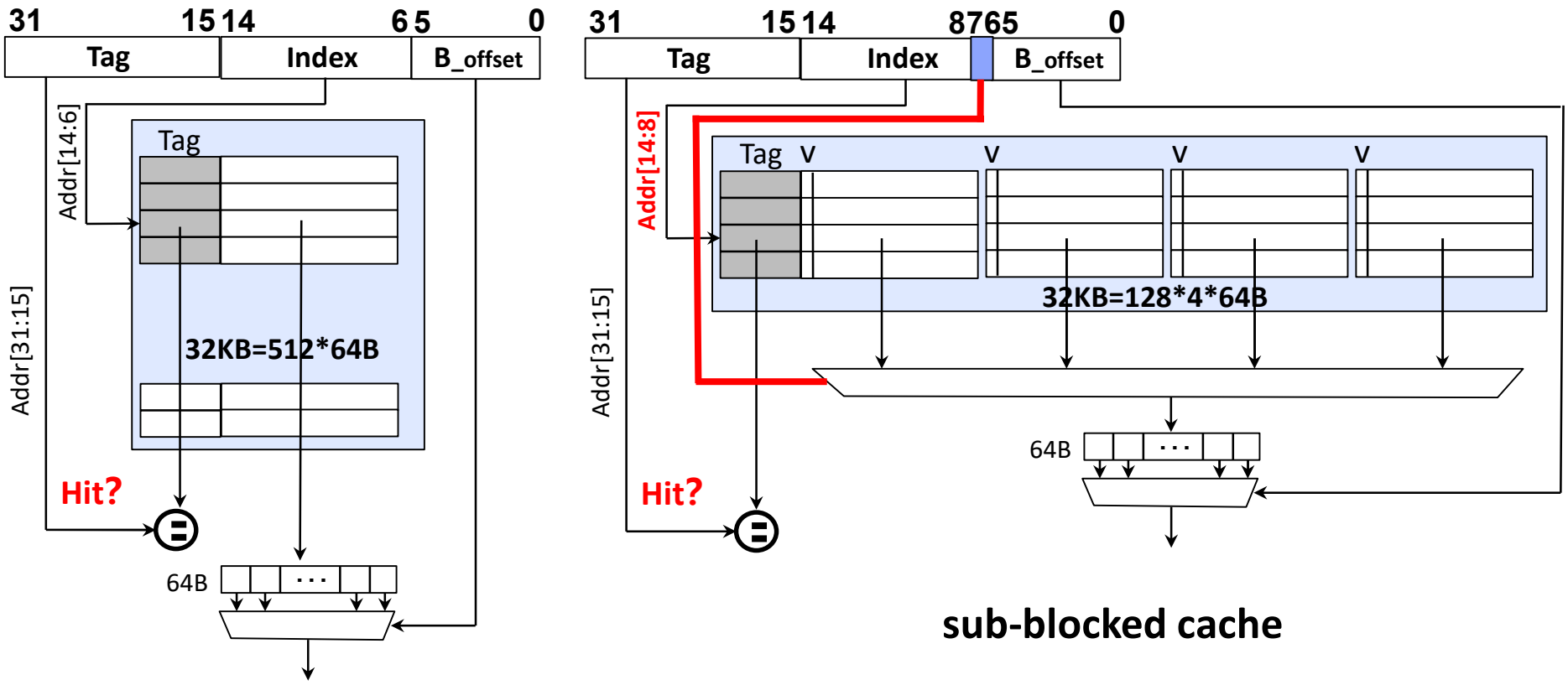
Faster solution: Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

Reducing Tag 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 (a.k.a 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?*



A single way of a sub-blocked cache



sub-blocked cache

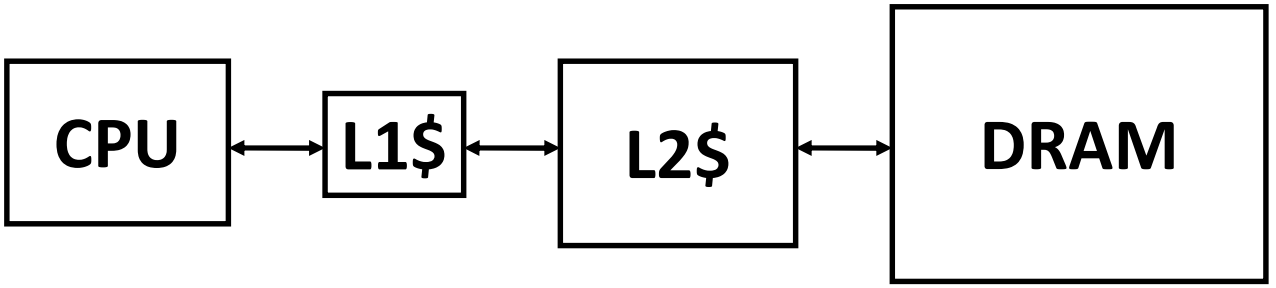
Reduce the % of storage for tags (+)

Good for false-sharing in multithreaded applications (+)

③Multilevel 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

③Multilevel Caches: Presence of L2 influences L1 design

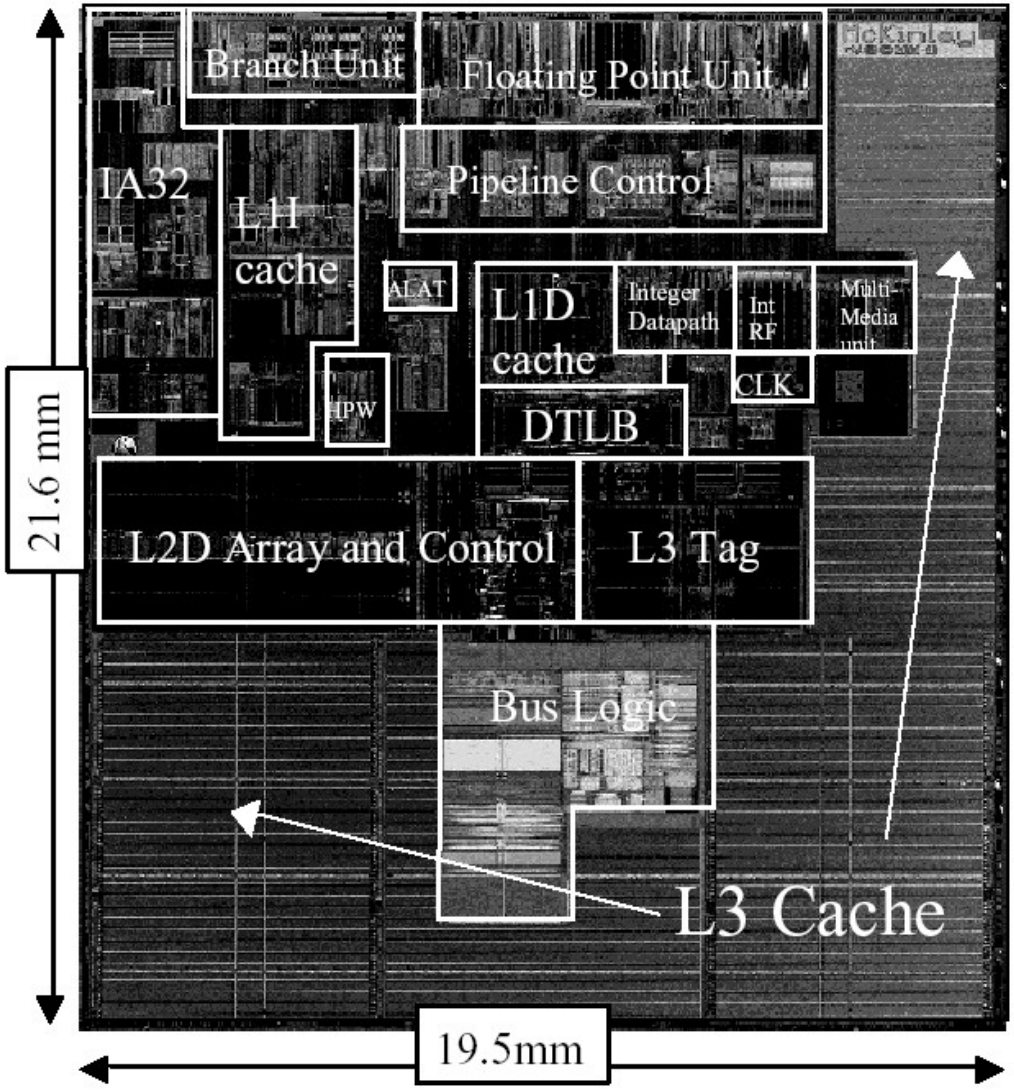
- Use smaller L1 if there is also L2
  - Trade increased L1 miss rate for reduced L1 hit time
  - Backup L2 reduces L1 miss penalty
  - Reduces average access energy
- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cache absorbs write traffic, doesn't go off-chip
  - At most one L1 miss 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)

③Multilevel Caches: Inclusion Policy

- **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 Caches
(Intel/HP, 2002)



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

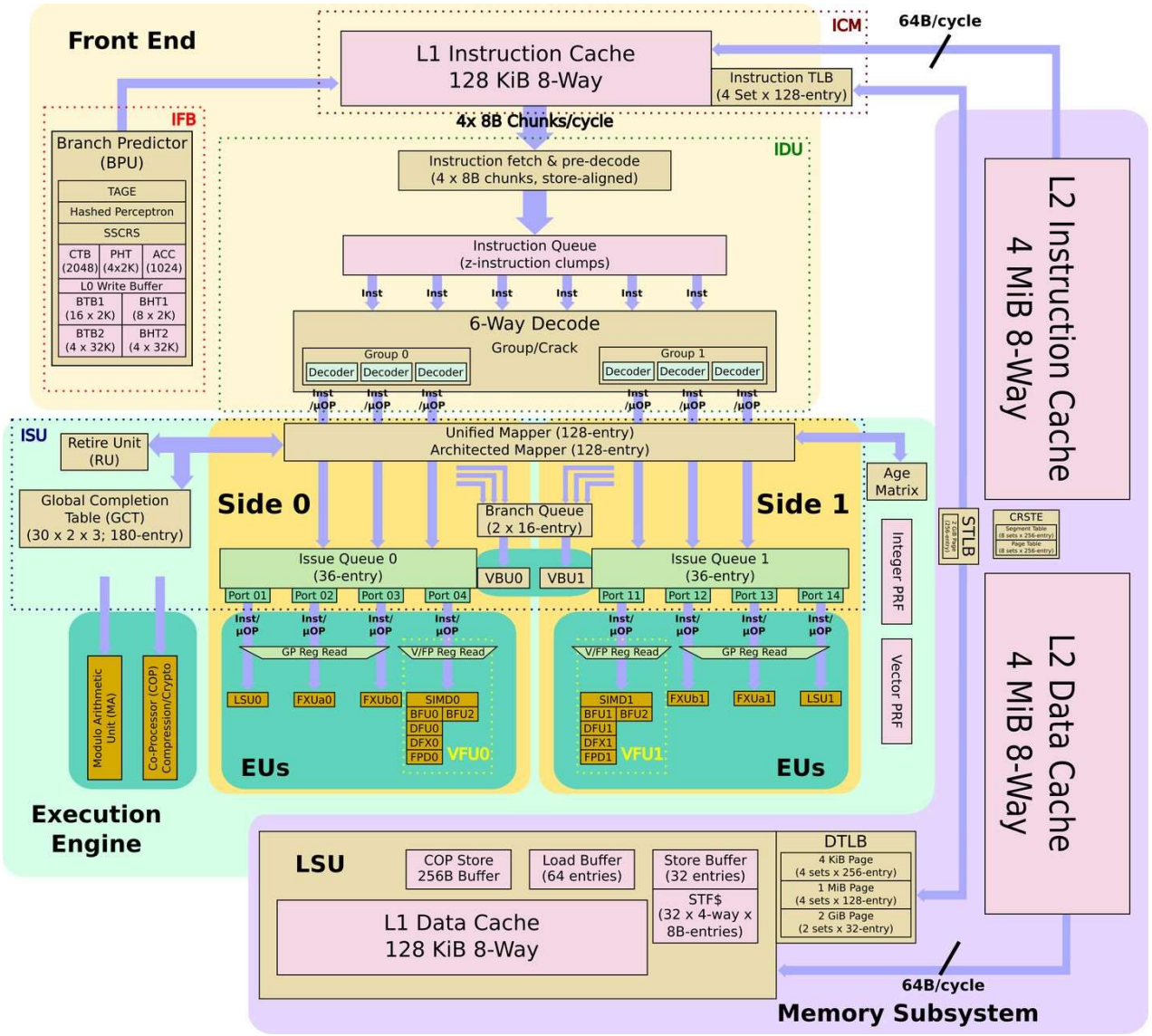
Level 3: 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

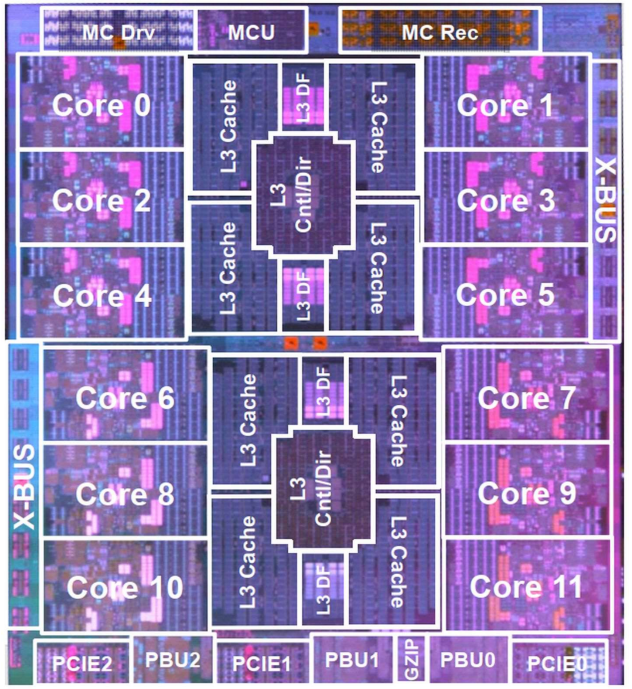
IBM z15 Mainframe Caches 2020

At ISSCC 2020 in San Francisco IBM mainframe chip details

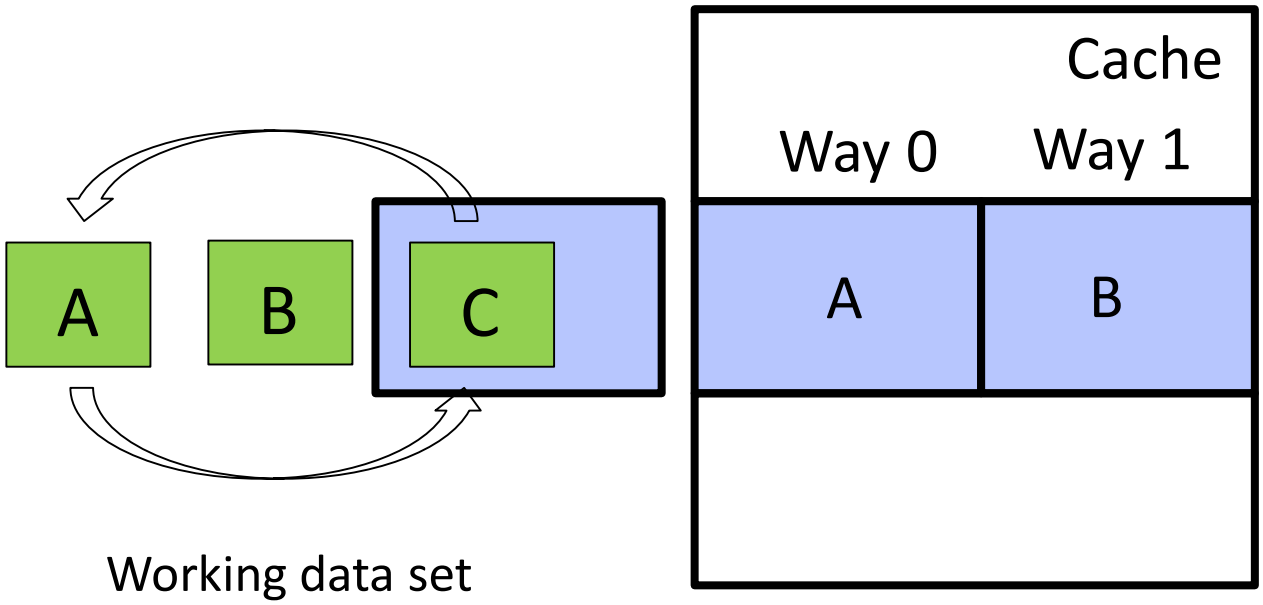
- z15 designed in 14nm FinFET technology with seventeen metal layers, 12.2 billion transistors/chip
- 12 cores/chip, 128KB L1 D&I cache, with 96MB L2 cache, 256MB L3 cache, and 960MB L4 off-chip cache.
- 5.2GHz clock rate, 4-way, 10-issue per cycle, 2 threads/core
- Up to 240 (190 for user, 60 for system management) processor chips in shared memory node

IBM z15 Mainframe Caches 2020

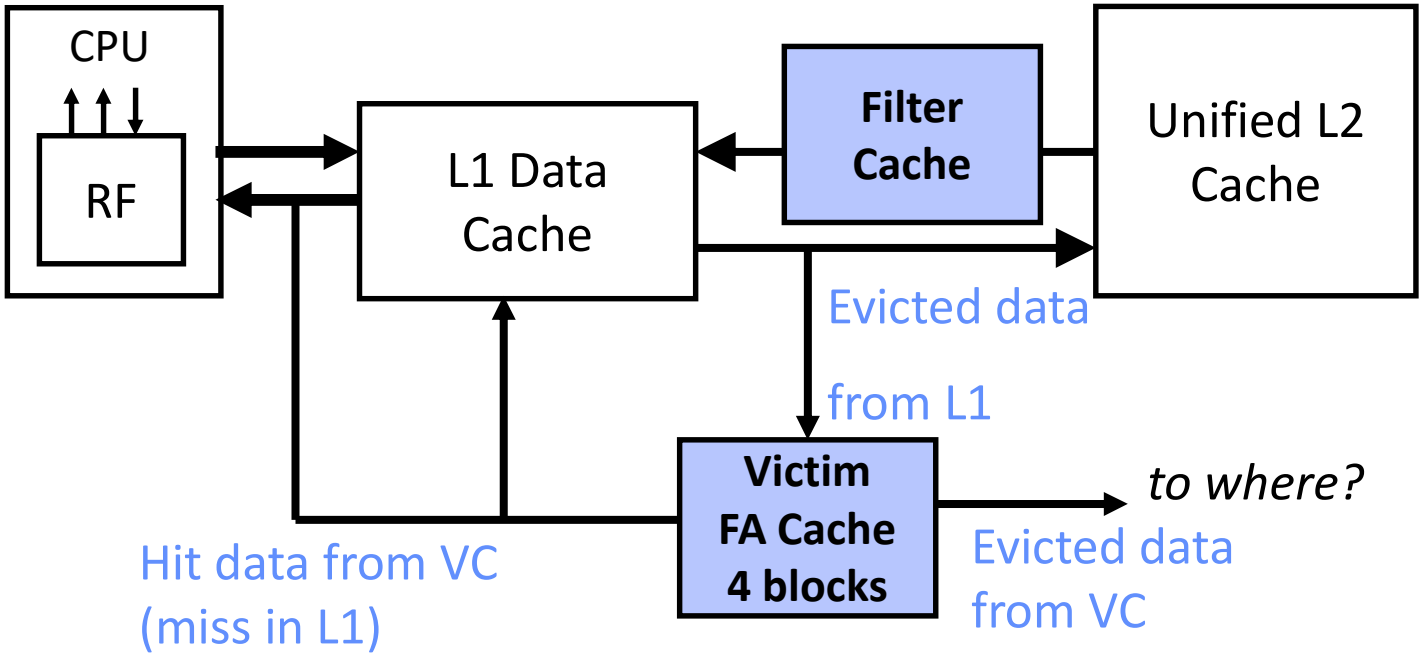




④Victim Caches: Motivation



④Victim Caches (HP 7200)



Victim cache is a small associative backup cache, added to a direct-mapped cache, which holds recently evicted lines

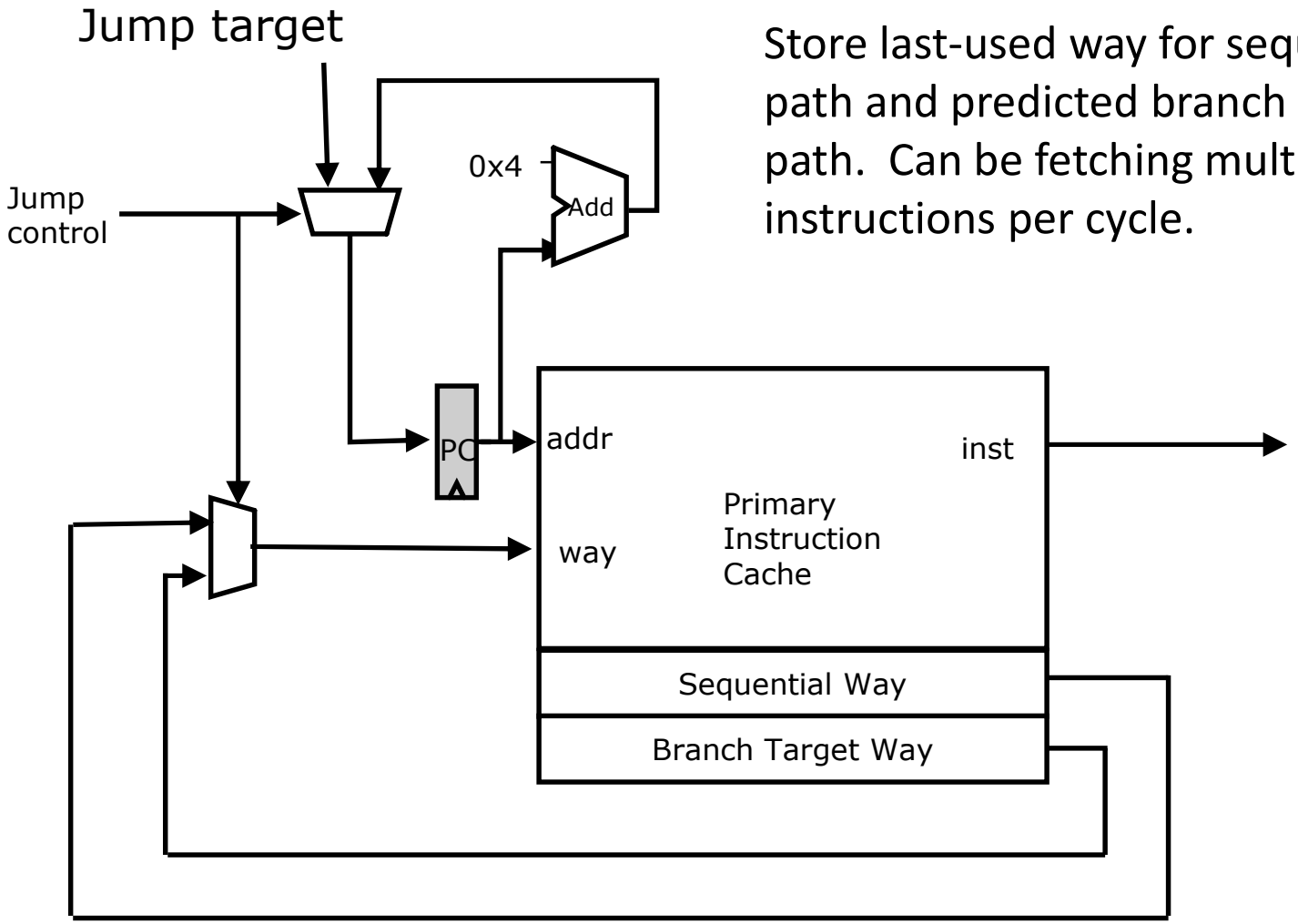
- First look up in direct-mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim->?

Fast hit time of direct mapped but with reduced conflict misses

⑤Prefetching

- 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?

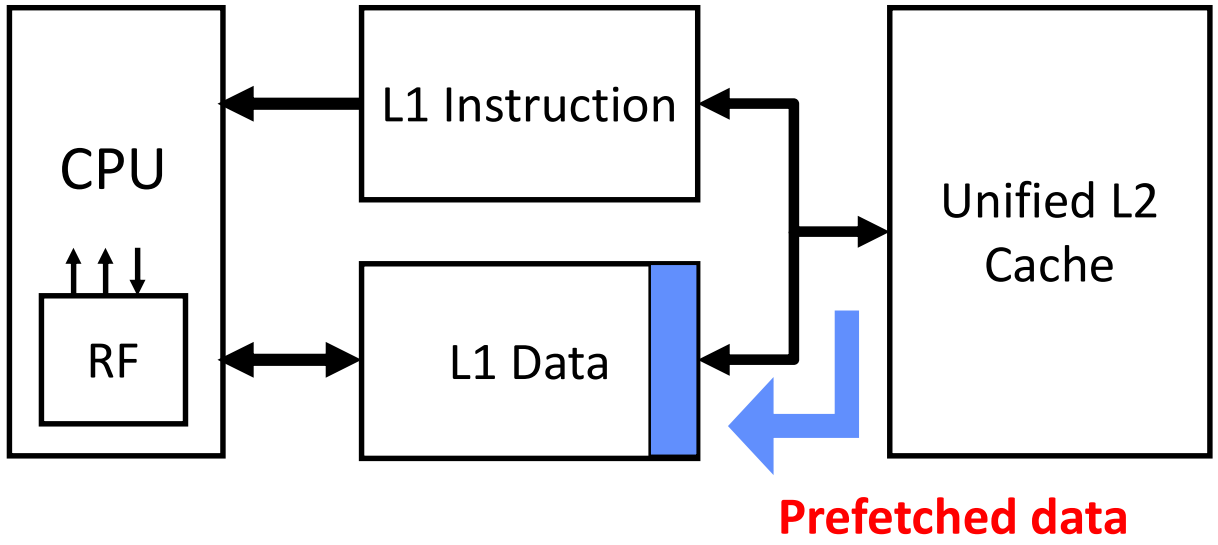
⑤Prefetching: Way-Predicting Instruction Cache (Alpha 21264-like)



Store last-used way for sequential path and predicted branch taken path. Can be fetching multiple instructions per cycle.

⑤Issues in Prefetching

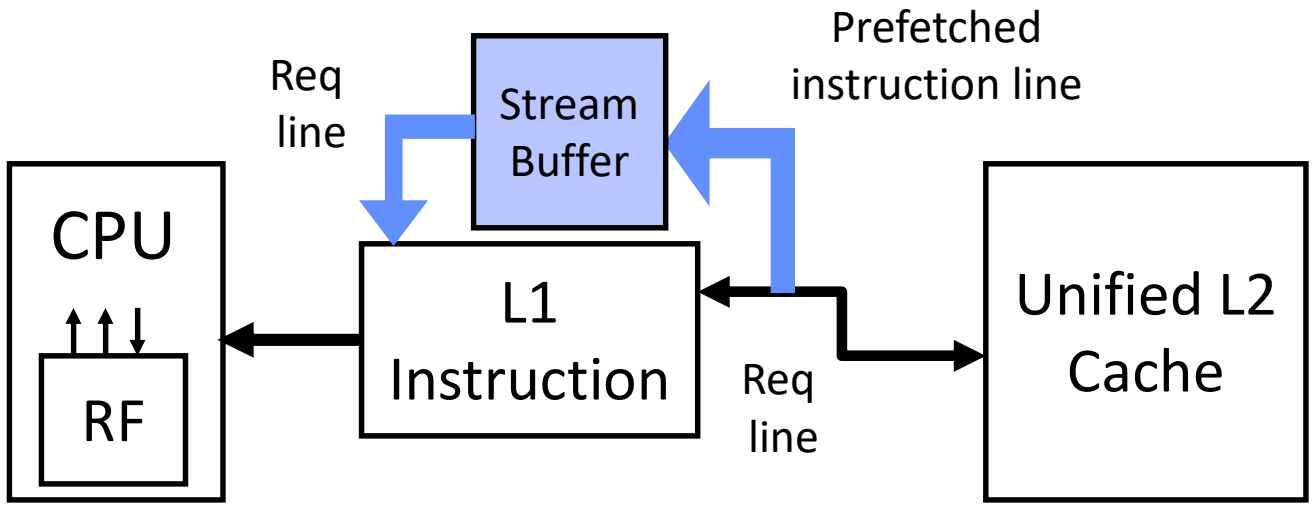
- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



⑤ Prefetching : Hardware Instruction Prefetching

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)



⑤Prefetching：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

⑤Prefetching：Software Prefetching

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

⑤Prefetching：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?*

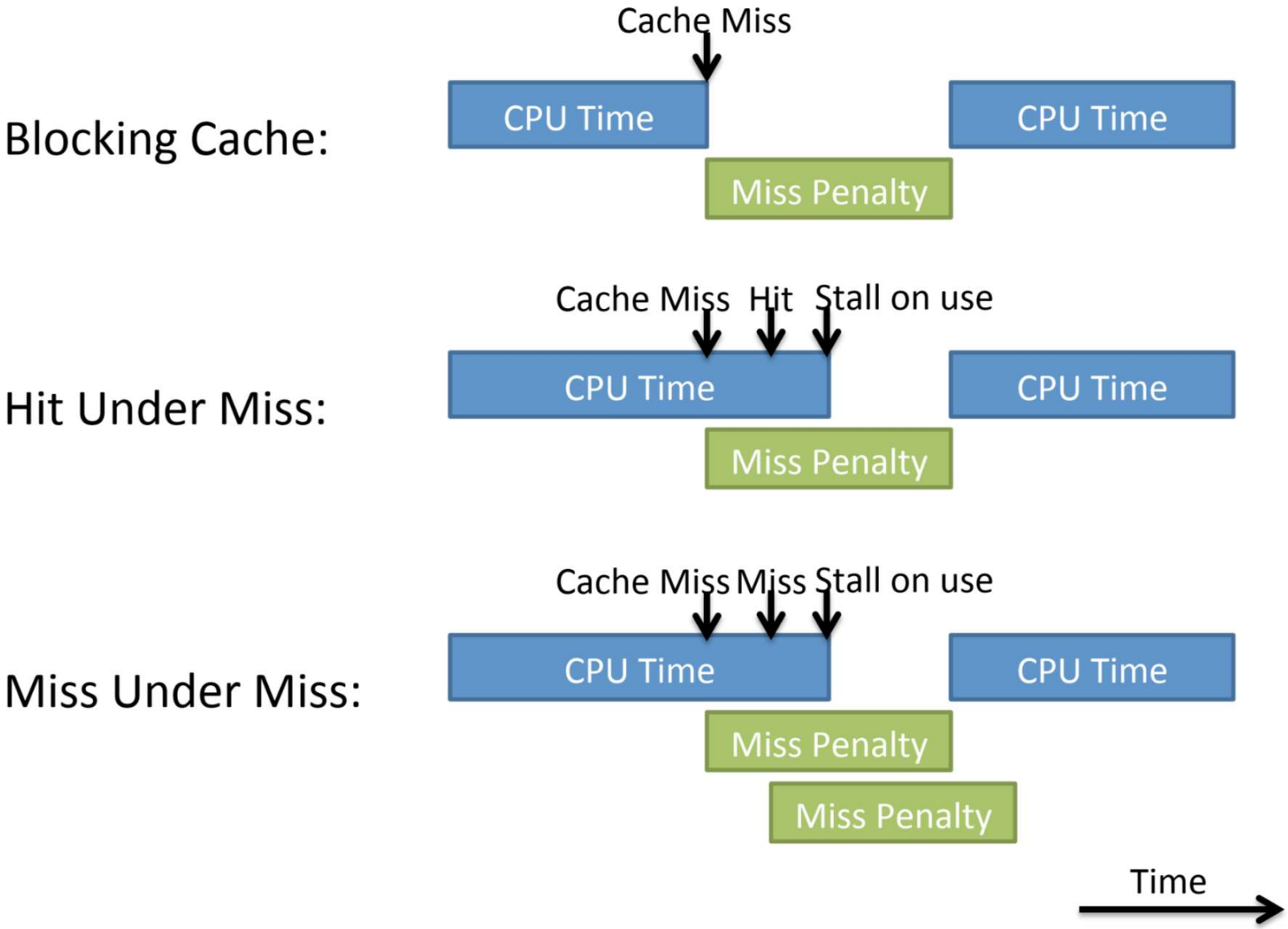
```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
 }
```

Must consider cost of prefetch instructions

⑥ Increasing Cache Bandwidth with Non-Blocking Caches(OOO)

- *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  - requires Full/Empty bits on registers or out-of-order execution
- "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses, and can get miss to line with outstanding miss (secondary miss)
  - Requires pipelined or banked memory system (otherwise cannot support multiple misses)
  - Pentium Pro allows 4 outstanding memory misses
  - Cray X1E vector supercomputer allows 2,048 outstanding memory misses

⑥ Non-Blocking Cache Timeline



Non-blocking Caches (a.k.a OOO Memory System, Lockup Free Caches)

Enable subsequent cache accesses after a cache miss has occurred

- Hit-under-miss
- Miss-under-miss (concurrent misses)

Suitable for in-order processor or OOO processors

Challenges

- Maintaining order when multiple misses that might return OOO
- Load or Store to an already pending miss address (need merge)

⑥ Non-Blocking Cache ：Miss Status Handling/holding Register (MSHR)/ Miss Address File (MAF)

MSHR/MAF

V	Block Address	Issued

V: Valid
Block Address: Address of cache block in memory system
Issued: Issued to Main Memory/Next level of cache

Load/Store Entry

V	MSHR Entry	Type	Offset	Destination

V: Valid
MSHR Entry: Entry Number
Type: {LW, SW, LH, SH, LB, SB}
Offset: Offset within the block
Destination: (Loads) Register, (Stores) Store buffer entry

⑥ Non-Blocking Cache Operation

On Cache Miss:

- Check MSHR for matched address
  - If found: Allocate new Load/Store entry pointing to MSHR
  - If not found: Allocate new MSHR entry and Load/Store entry
  - If all entries full in MSHR or Load/Store entry table, stall or prevent new LDs/STs

On Data Return from Memory:

- Find Load or Store waiting for it
  - Forward Load data to processor/Clear Store Buffer
  - Could be multiple Loads and Stores
- Write Data to Cache

When Cache Lines is Completely Returned:

- De-allocate MSHR entry

⑥ Non-Blocking Cache with In-order Pipelines

Need Scoreboard for Individual Registers

On Load Miss:

- Mark Destination Register as Busy

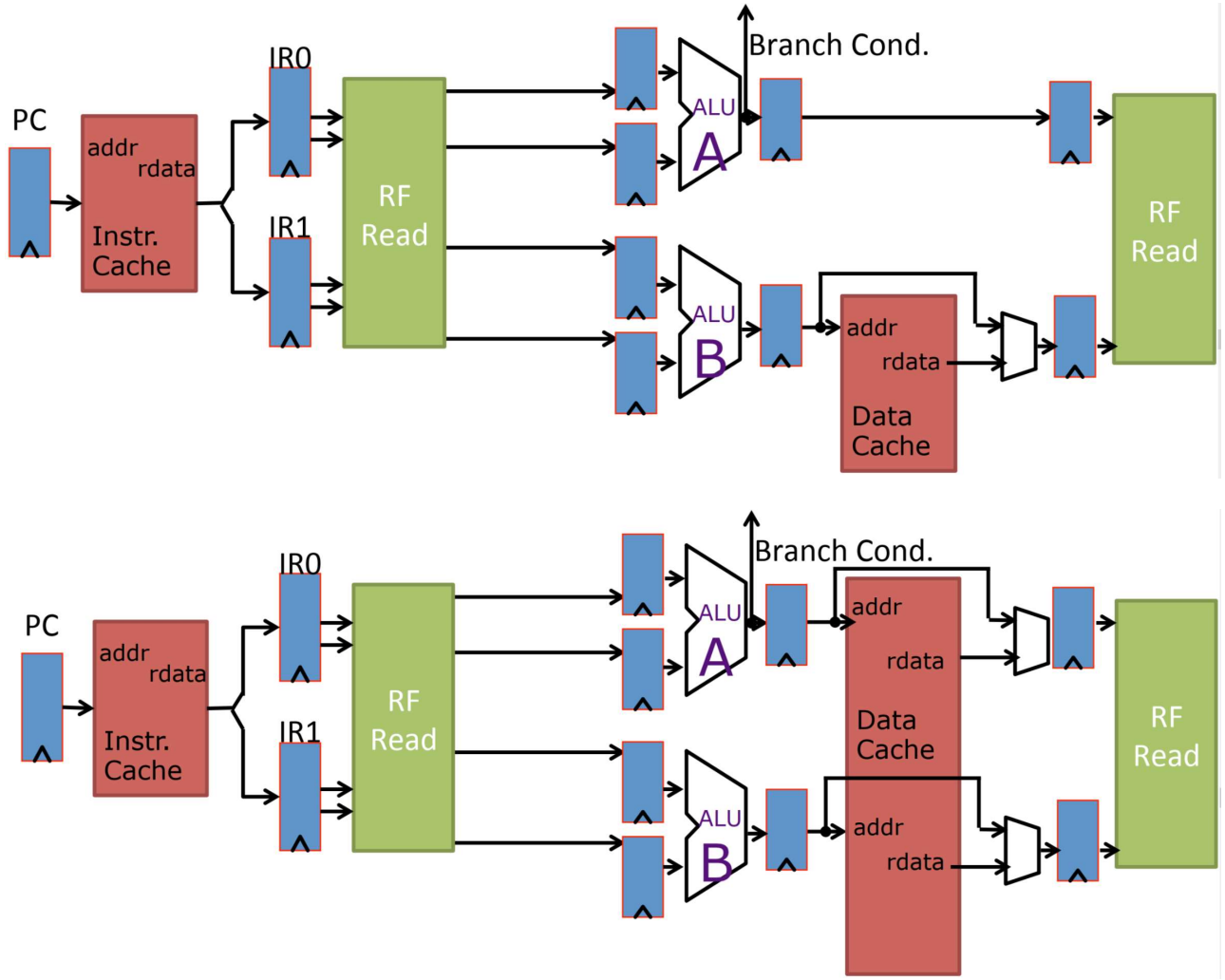
On Load Data Return:

- Mark Destination Register as Available

On Use of Busy Register:

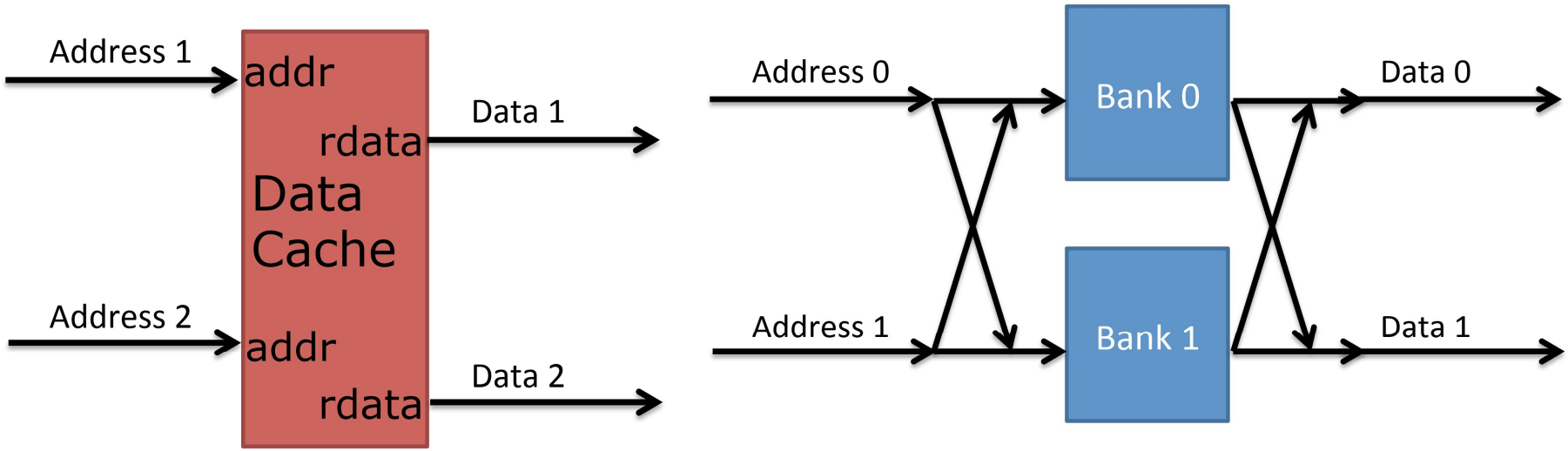
- Stall Processor

⑦ Increasing Cache Bandwidth Multiporing and Banking



Challenges: Two stores to the same line,
or load and store to same line

⑦ Multiport Caches



Ture Multiport Caches:

- Large area increase (could be double for 2-port)
- Hit time increase (can be made small)

Banked Caches: Partition Address Space into multiple banks – use portions of address (low or high order interleaved)
Benefits: Higher throughput
Challenges: Bank Conflicts & Extra Wiring

Agenda

Advanced Cache Optimizations

① Pipelined Cache Write

② Write Buffer

③ Multilevel Caches

④ Victim Caches

⑤ Prefetching(hardware/software)

⑥ Non-Blocking Cache

⑦ Multiporting and Banking

⑧ Software Optimizations

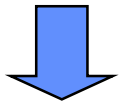
⑨ Critical Word First/Early Restart

⑧ Software Optimizations: 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

⑧ Software Optimizations: 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?

⑧ Software Optimizations: Loop Fusion

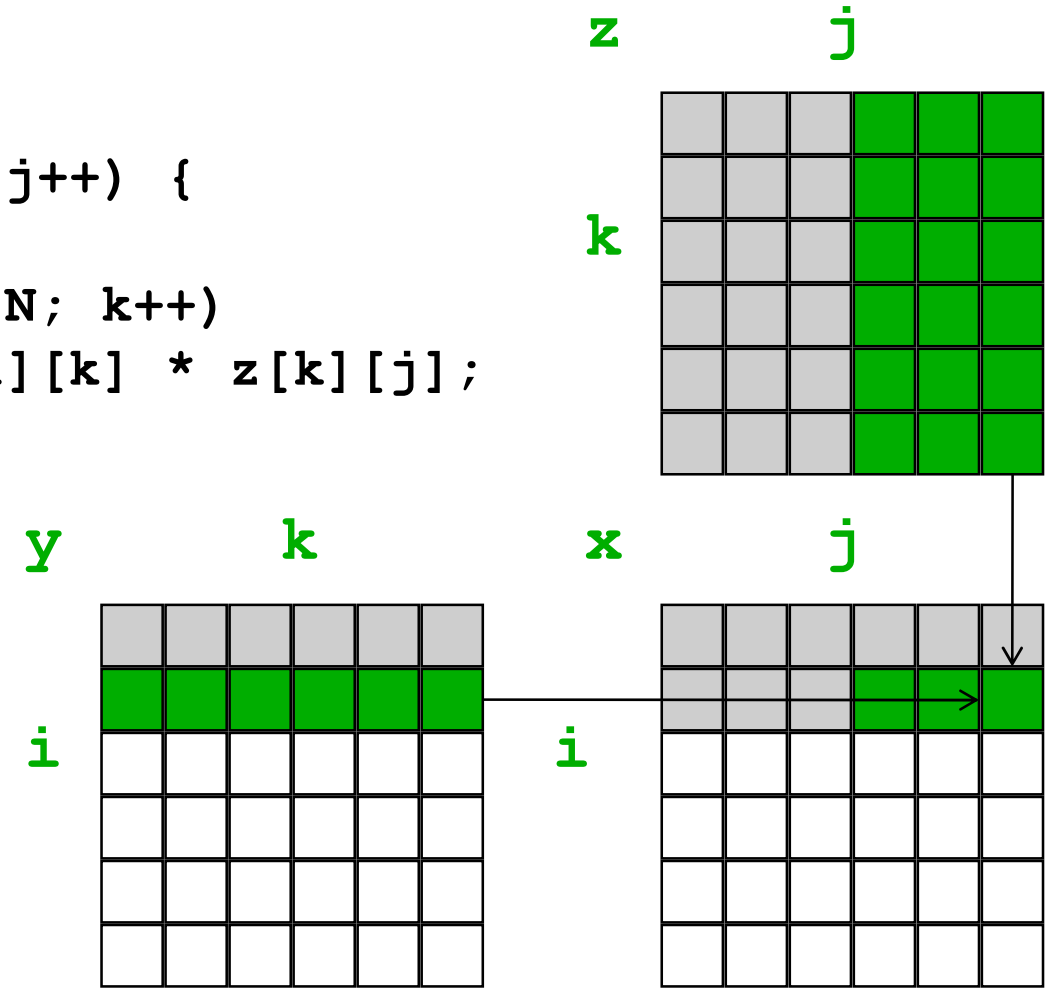
```
for(i=0; i < N; i++)
      a[i] = b[i] * c[i];

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?

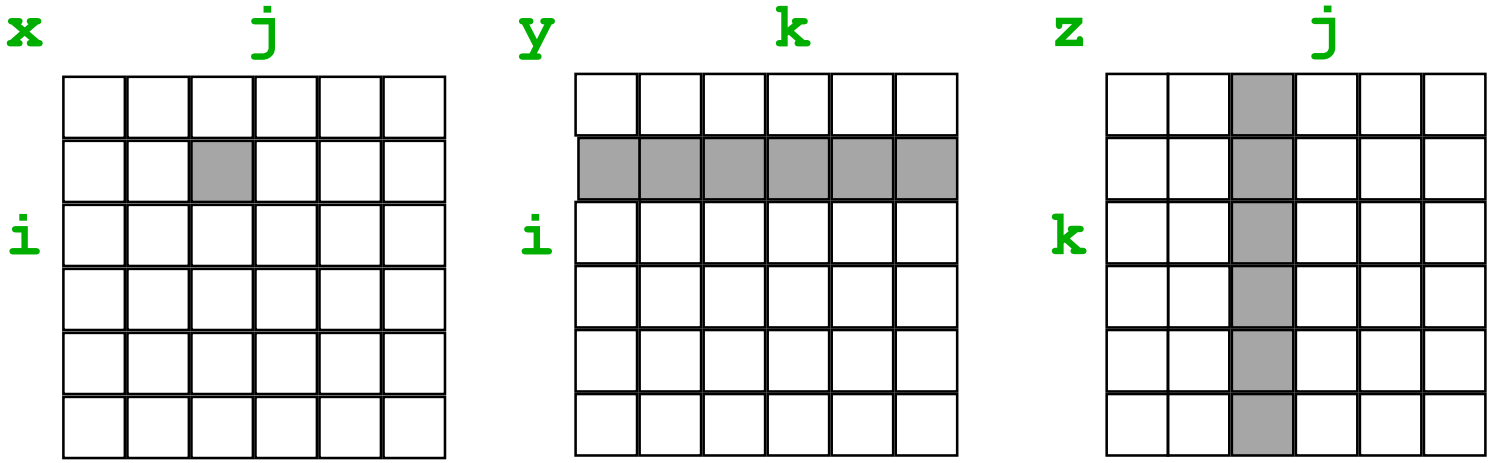
⑧ Software Optimizations: Matrix Multiply, Naïve Code

```
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*

⑧ Software Optimizations: Matrix Multiply, Naïve Code (If there is no Cache)



For each element of X, read one row of Y and one column of Z

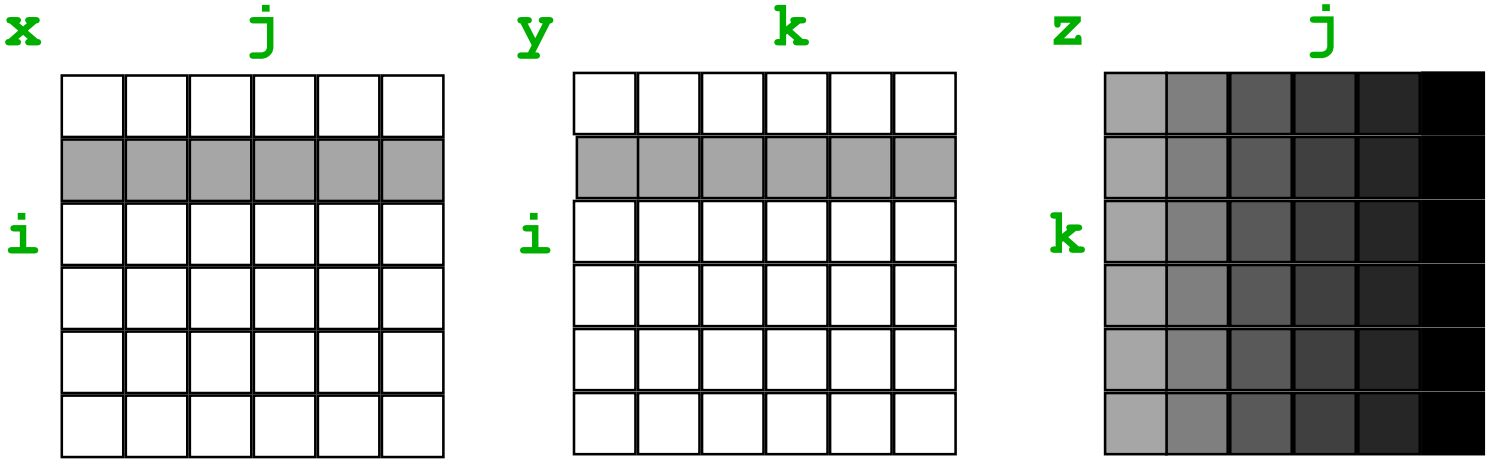
Total Mem access = N^2 x **(2 +** N **+** N**) =** $2\mathrm{N}^2 + 2\mathrm{N}^3$

Computational intensity : $2\mathrm{N}^3/(2\mathrm{N}^2 + 2\mathrm{N}^3) \approx 1$

(including N^3**multiplies and** N^3**addition)**

⑧ Software Optimizations: Matrix Multiply, Naïve Code (If Cache size is 3N)

```
for(i=0; i < N; i++)
        【read row i of y into fast Mem】
      for(j=0; j < N; j++) {
         【read x[i][j] into fast Mem】
         【read column j of z into fast Mem】
         r = 0;
         for(k=0; k < N; k++)
           r = r + y[i][k] * z[k][j];
         x[i][j] = r;
         【write x[i][j] back to fast Mem】
      }
```



For each row of X, read one row of Y and every column of Z

⑧ Software Optimizations: Matrix Multiply, Naïve Code (If Cache size is 3N)

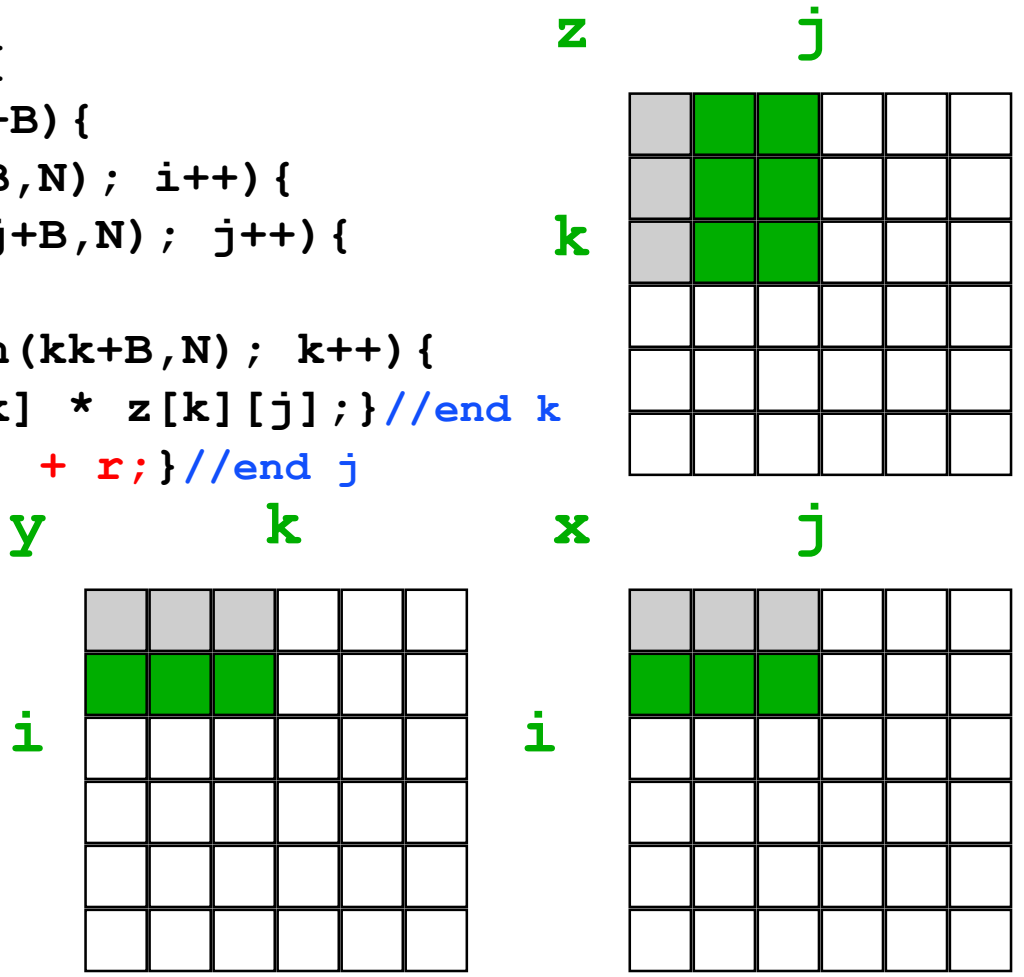
```
for(i=0; i < N; i++)
        【read row i of y into fast Mem】
      for(j=0; j < N; j++) {
         【read x[i][j] into fast Mem】
         【read column j of z into fast Mem】
         r = 0;
         for(k=0; k < N; k++)
           r = r + y[i][k] * z[k][j];
         x[i][j] = r;
         【write x[i][j] back to fast Mem】
      }
```

Total Mem access = N^3 to read each column of z N^2 times $(N * N^2)$
+ N^2 to read each row of y once $(N * N)$
+ $2N^2$ to read and write each element of x $(N^2 + N^2)$
= $N^3 + 3N^2$

Computational intensity : $2N^3/(N^3 + 3N^2) \approx 2$

⑧ Software Optimizations: Matrix Multiply with Cache Tiling
(If Cache size is bigger than $3B^2$)

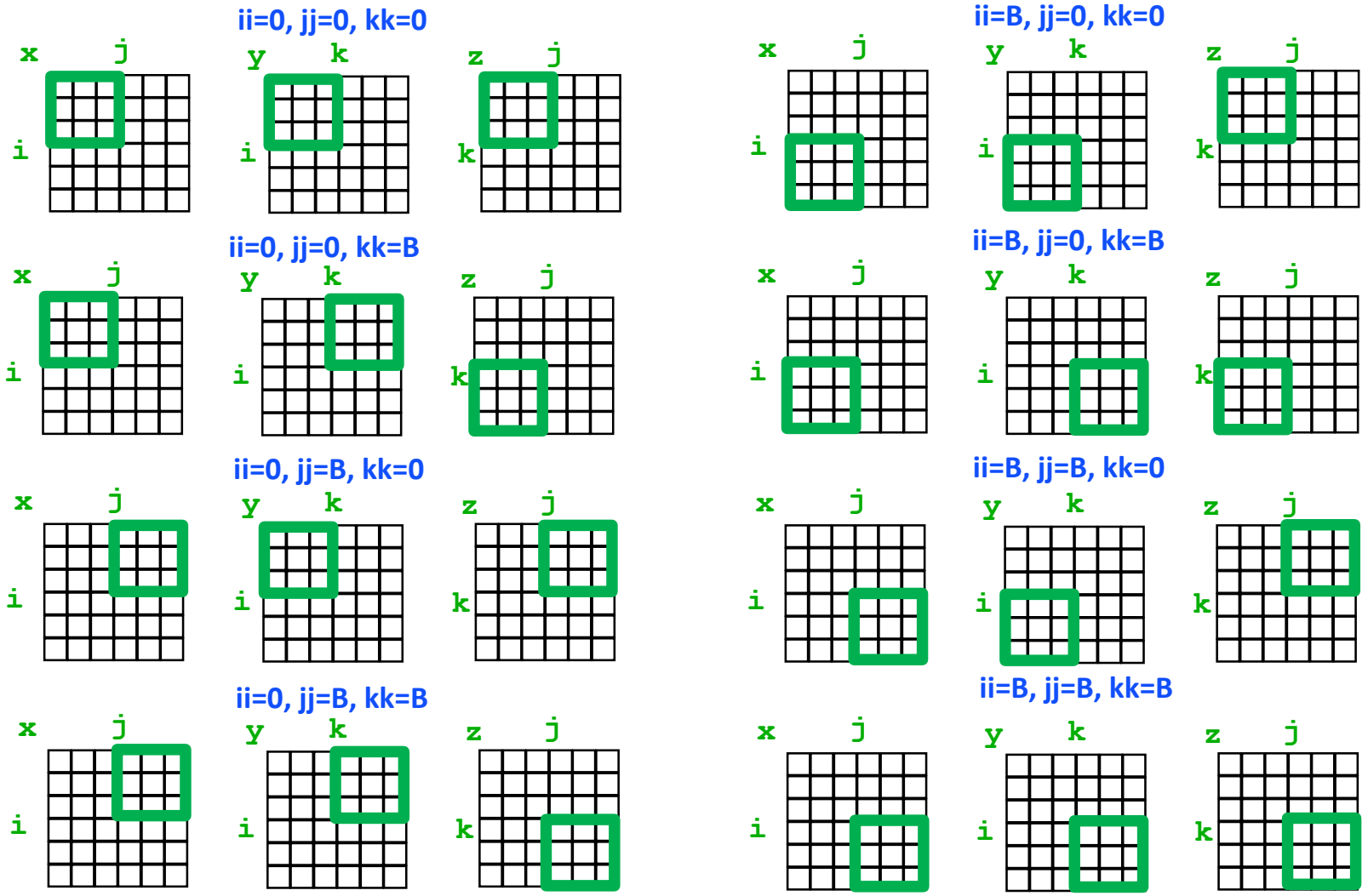
```
for(ii=0; ii < N; ii=ii+B){
  for(jj=0; jj < N; jj=jj+B){
     for(kk=0; kk < N; kk=kk+B){
       for(i=ii; i < min(ii+B,N); i++){
         for(j=jj; j < min(jj+B,N); j++){
            r = 0;
            for(k=kk; k < min(kk+B,N); k++){
               r = r + y[i][k] * z[k][j];}//end k
         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



What type of locality does this improve?

⑧ Software Optimizations: Matrix Multiply with Cache Tiling

(If Cache size is bigger than $3B^2$)



Total Mem access = $\left(2\left(\frac{N}{B}\right)^2 + \left(\frac{N}{B}\right)^2 \text{x } \left(\frac{N}{B}\right) + \left(\frac{N}{B}\right)^2 \text{x } \left(\frac{N}{B}\right)\right) \text{x } B^2 = 2N^2 + \frac{2N^3}{B}$

⑧ Software Optimizations: Matrix Multiply with Cache Tiling (If Cache size is bigger than $3B^2$)

```
for(ii=0; ii < N; ii=ii+B){
   for(jj=0; jj < N; jj=jj+B){
     【read B*B block of x into fast Mem】
     for(kk=0; kk < N; kk=kk+B){
        【read B*B block of y into fast Mem】
        【read B*B block of z into fast Mem】
       for(i=ii; i < min (ii+B,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;}
           }
       }
   }
```

The larger the block size, the more efficient our algorithm will be, however all three blocks from x,y,z must fit in Cache

$3b^2 \le M_{fast}$, so $b \le (M_{fast}/3)^{1/2}$

Total Mem access = N^3 /B to read each block of z $(\frac{N}{B})^3$ times ($(\frac{N}{B})^3 * B^2 = N^3$ /B)

+ N^3 /B to read each block of y $(\frac{N}{B})^3$ times

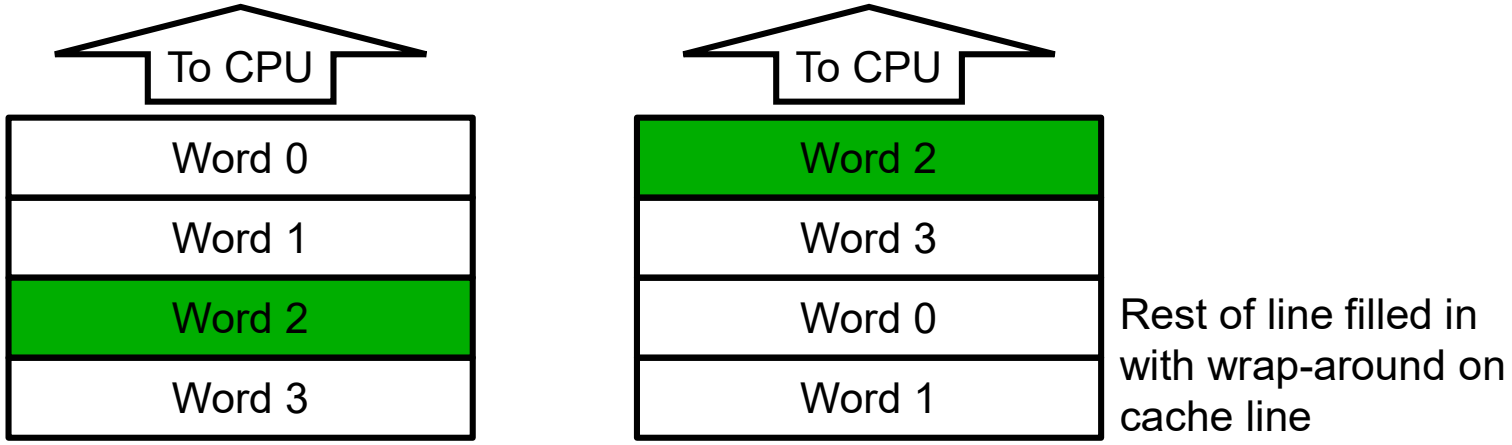
+ $2N^2$ read and write each block of x once ($(\frac{N}{B})^2 * B^2 = N^2$)

= $2N^3$ /B + $2N^2$

Computational intensity : $2N^3/(2N^3$ /B + $2N^2) \approx B$ when N is big

Reduce Miss Penalty of Long Blocks: Early Restart and Critical Word First

- Don't wait for full block before restarting CPU
- *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
  - Long blocks more popular today ⇒ Critical Word 1st Widely used



⑨ Critical Word First

Request the missed word from memory first

Rest of Cache line comes after “critical word”

- Commonly words come back in rotated order

Basic Blocking Cache:

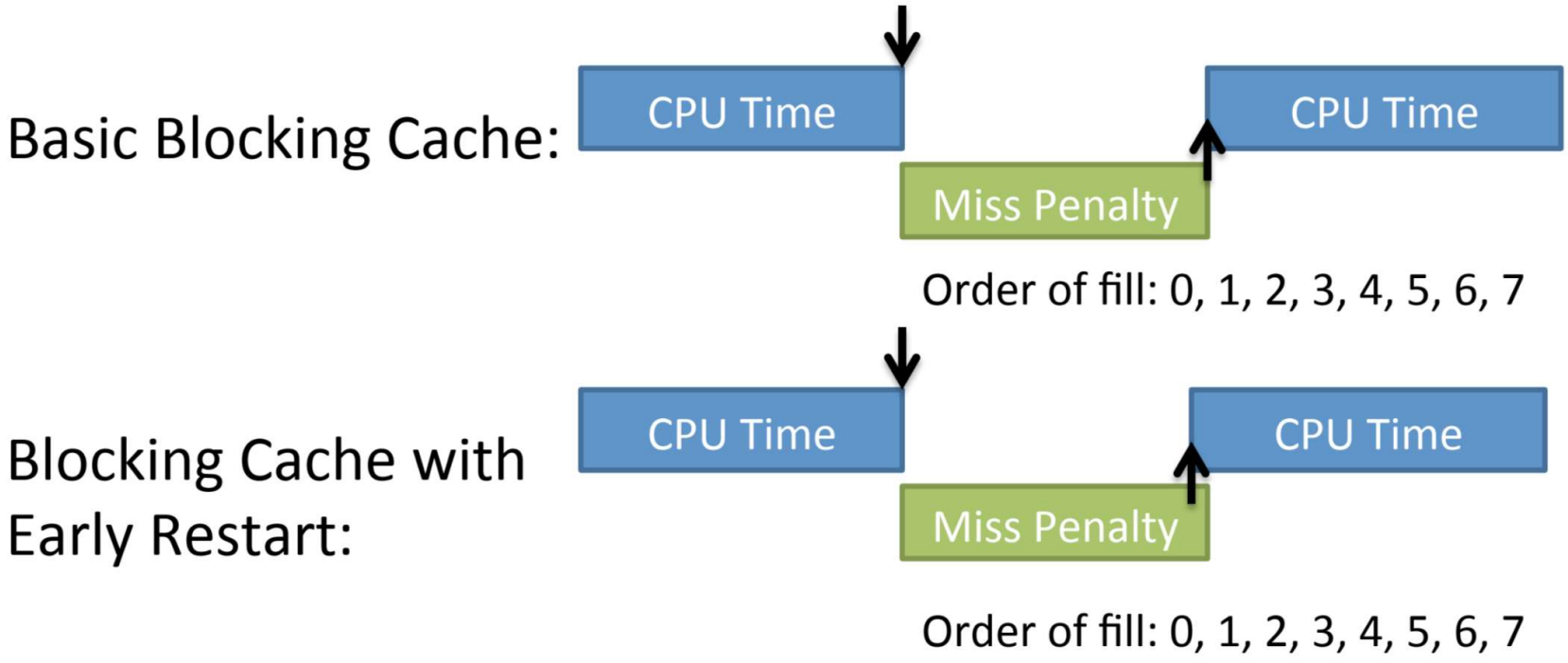
Order of fill: 0, 1, 2, 3, 4, 5, 6, 7

Blocking Cache with Critical Word first:

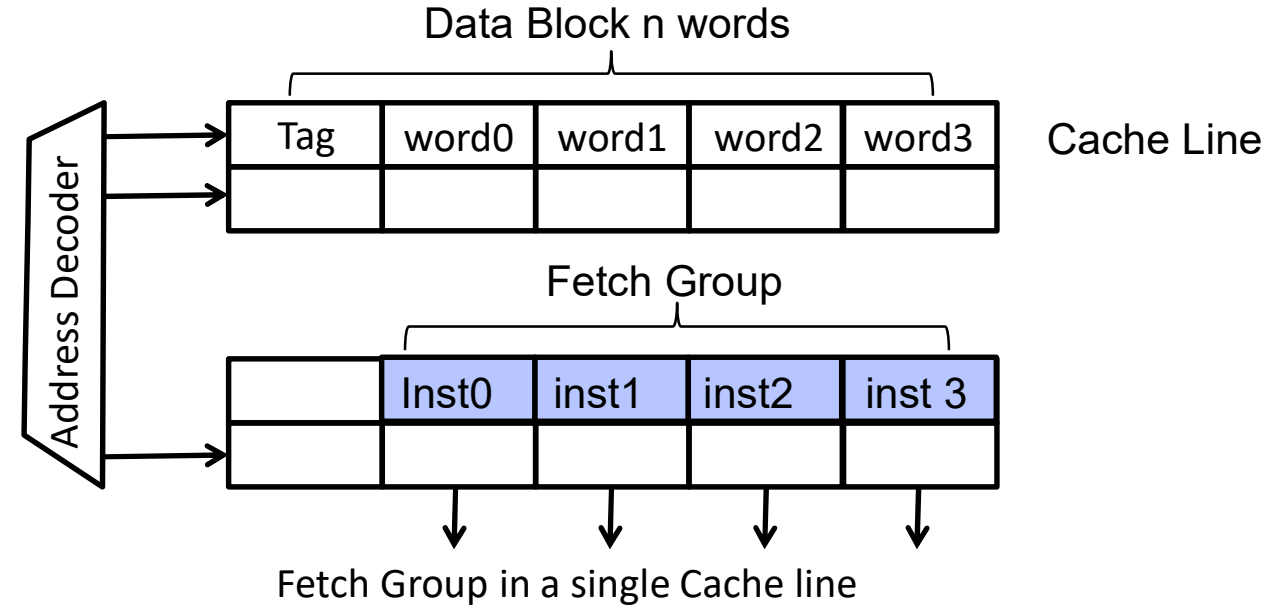
Order of fill: 3, 4, 5, 6, 7, 0, 1, 2

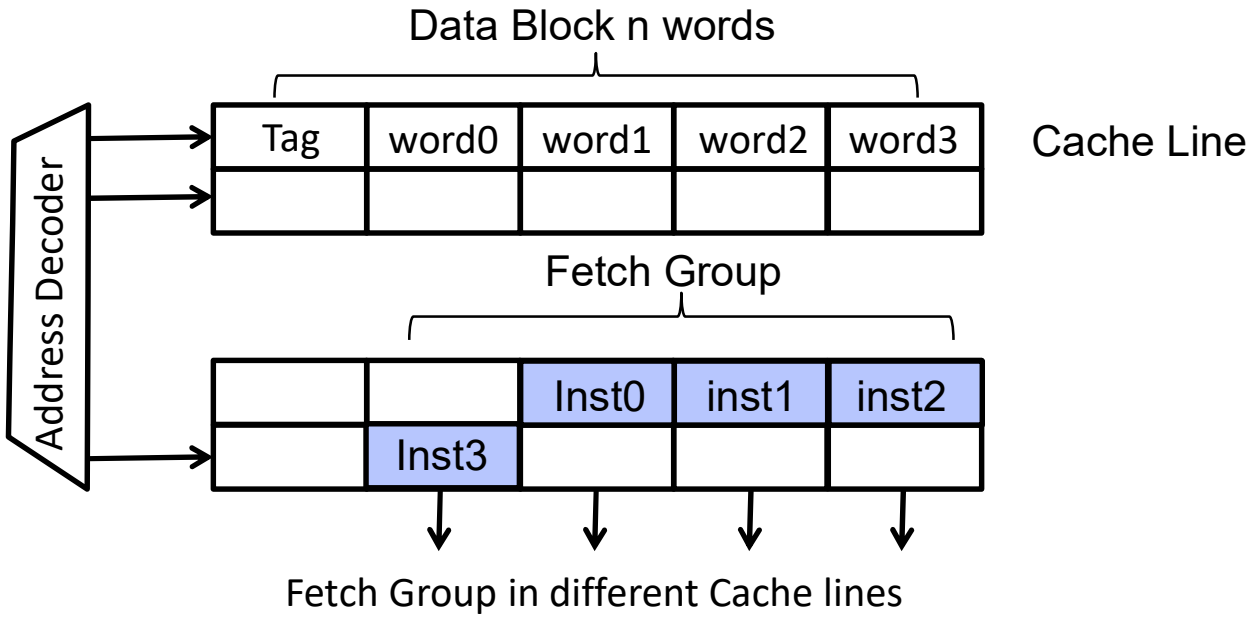
⑨ Critical Word First：Early Restart

Data returns from memory in order
Processor Restarts when needed word is returned



One more thing: I-Cache for n-way superscalar





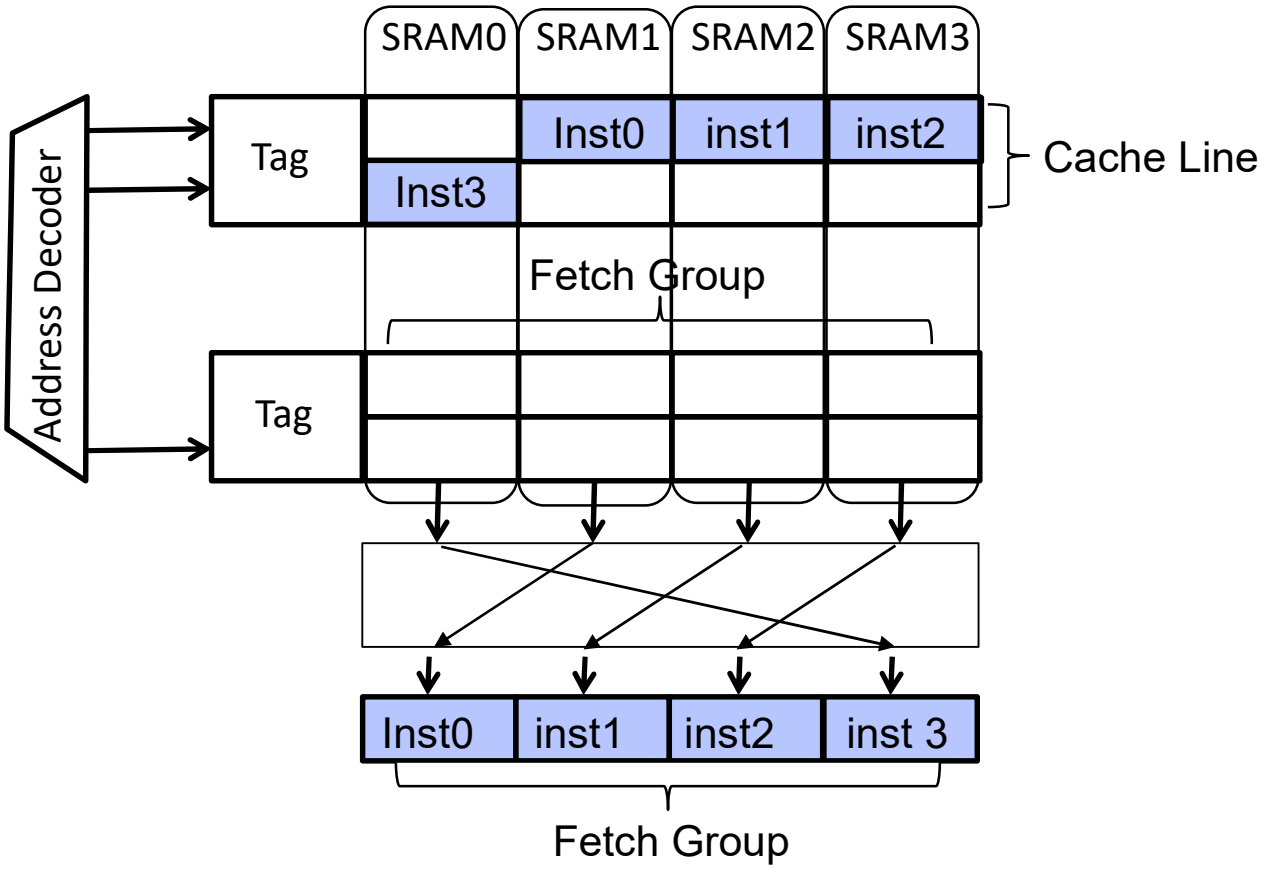
Recap: Fetch Logic and Alignment

Cycle	Addr	Instr
0	0x000	OpA
0	0x004	OpB
1	0x008	OpC
1	0x00C	J 0x100
…		
2	0x100	OpD
2	0x104	J 0x204
…		
3	0x204	OpE
3	0x208	J 0x30C
…		
4	0x30C	OpF
4	0x310	OpG
5	0x314	OpH

0x000	0	0	1	1
…				
0x100	2	2		
…				
0x200		3	3	
…				
0x300				4
0x310	4	5		

Fetching across cache lines is very expensive (need extra ports)

I-Cache for Fetch Alignment



Next Lecture：Address Translation & Virtual Memory

(Memory System)

Acknowledgements

- Some slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - David Patterson (UCB)
  - David Wentzlaff (Princeton University)

- MIT material derived from course 6.823
- UCB material derived from course CS252 and CS 61C

A single way of a sub-blocked cache