

COM515 Advanced Computer Architecture

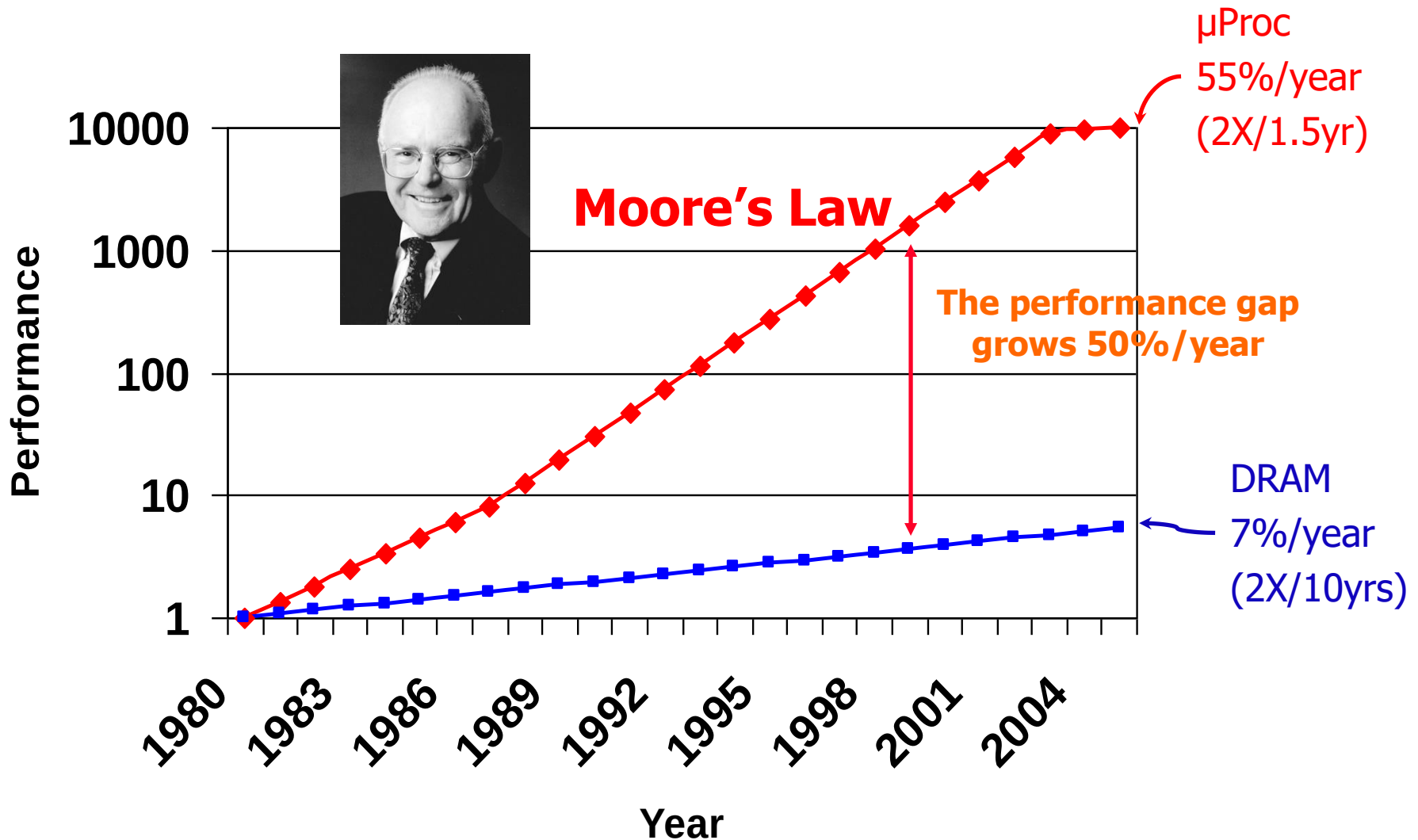
Lecture 8. Memory Hierarchy Design I

Prof. Taeweon Suh

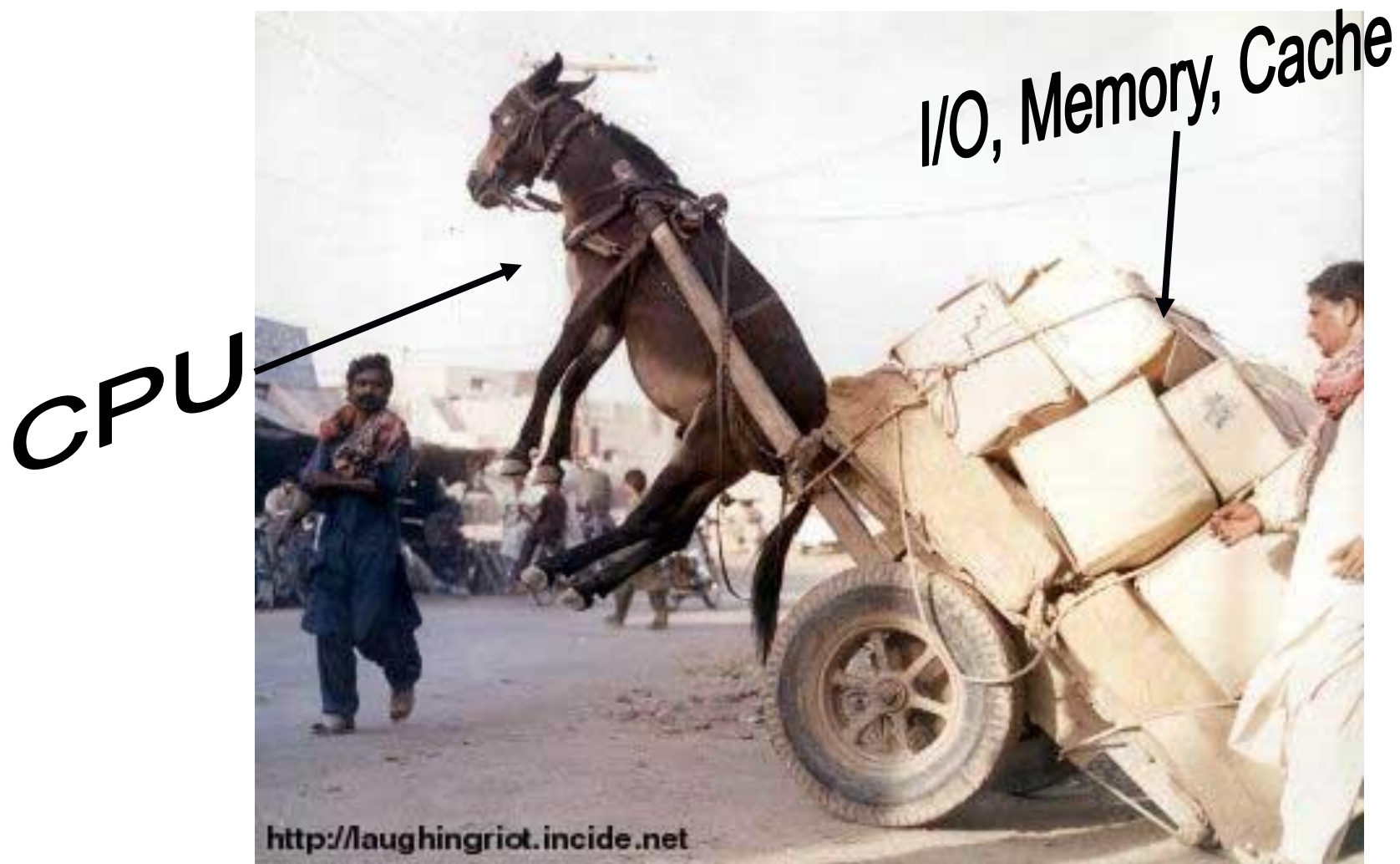
Computer Science Education

Korea University

CPU vs Memory Performance



An Unbalanced System

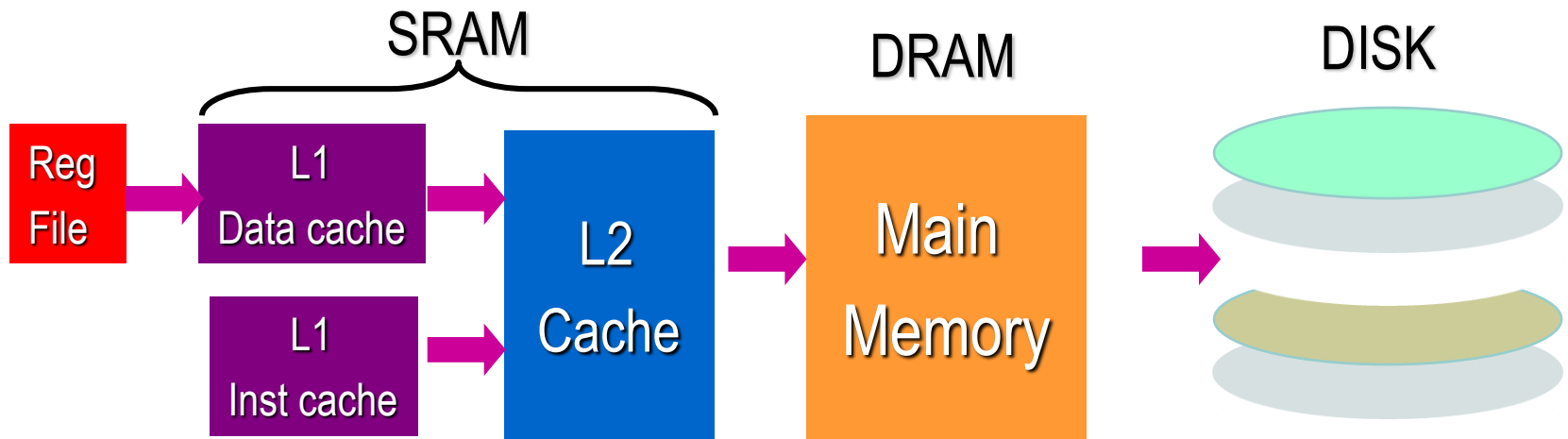


Source: Bob Colwell keynote ISCA'29 2002

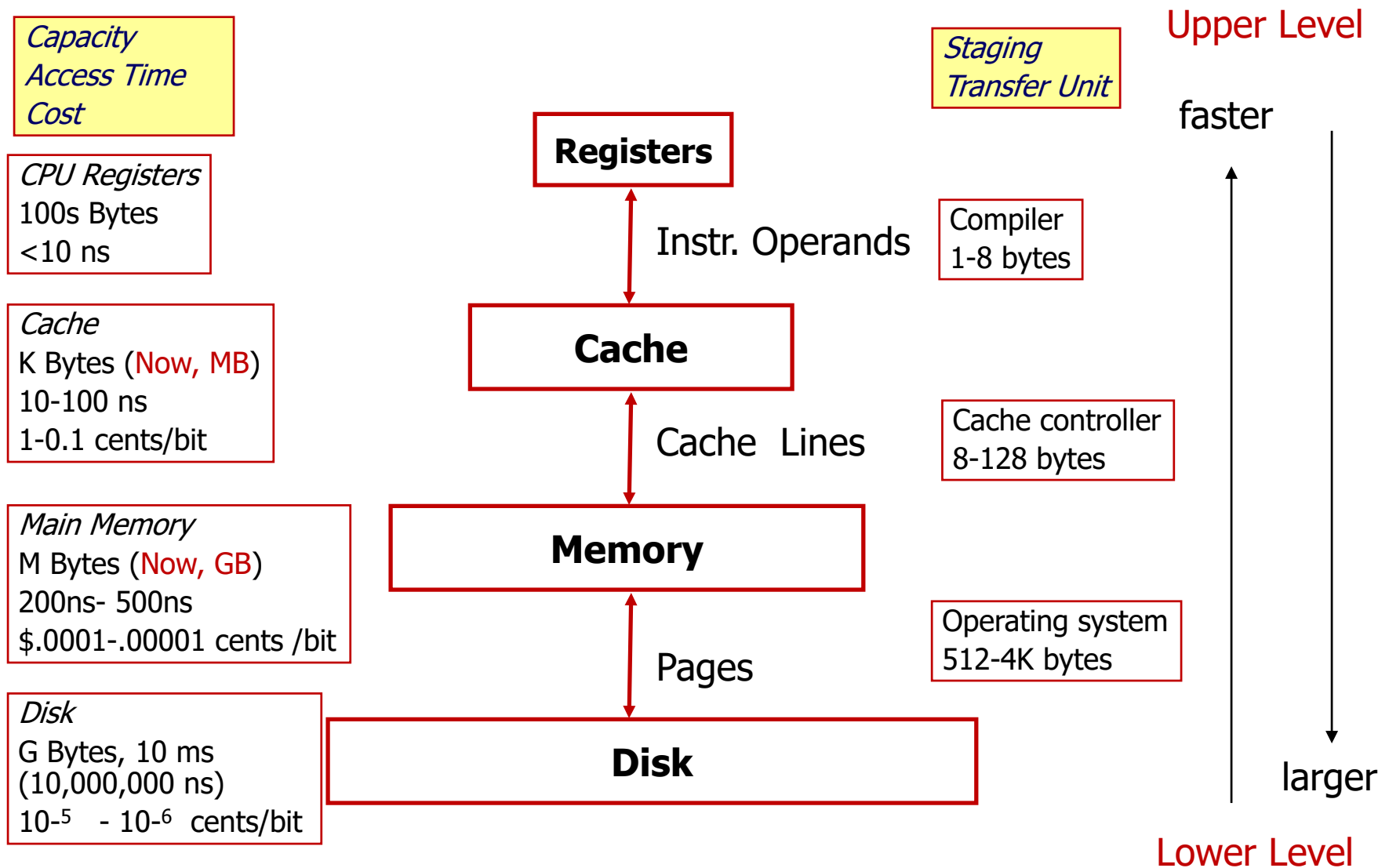
Memory Issues

- Latency
 - Time to move through the longest circuit path (from the start of request to the response)
- Bandwidth
 - Number of bits transported at one time
- Capacity
 - Size of memory
- Energy
 - Cost of accessing memory (to read and write)

Model of Memory Hierarchy



Levels of Memory Hierarchy



Topics covered

- Why do caches work
 - Principle of program locality
- Cache hierarchy
 - Average memory access time (AMAT)
- Types of caches
 - Direct mapped
 - Set-associative
 - Fully associative
- Cache policies
 - Write back vs. write through
 - Write allocate vs. No write allocate

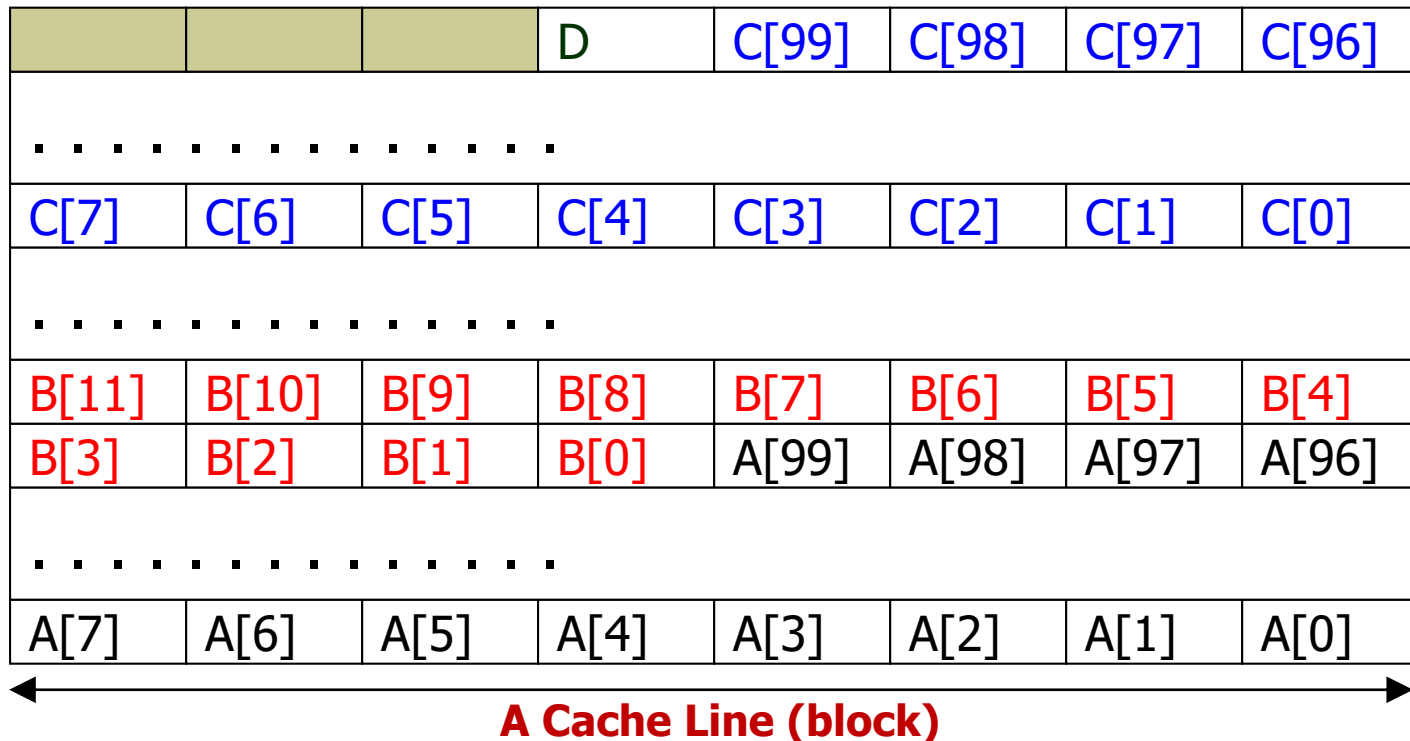
Why Caches Work?

- The size of cache is tiny compared to main memory
 - How to make sure that the data CPU is going to access is in caches?
- Caches take advantage of the principle of locality in your program
 - **Temporal Locality** (locality in time)
 - If a memory location is referenced, then it will tend to be referenced again soon. So, keep **most recently accessed** data items closer to the processor
 - **Spatial Locality** (locality in space)
 - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon. So, move blocks consisting of **contiguous words** closer to the processor

Example of Locality

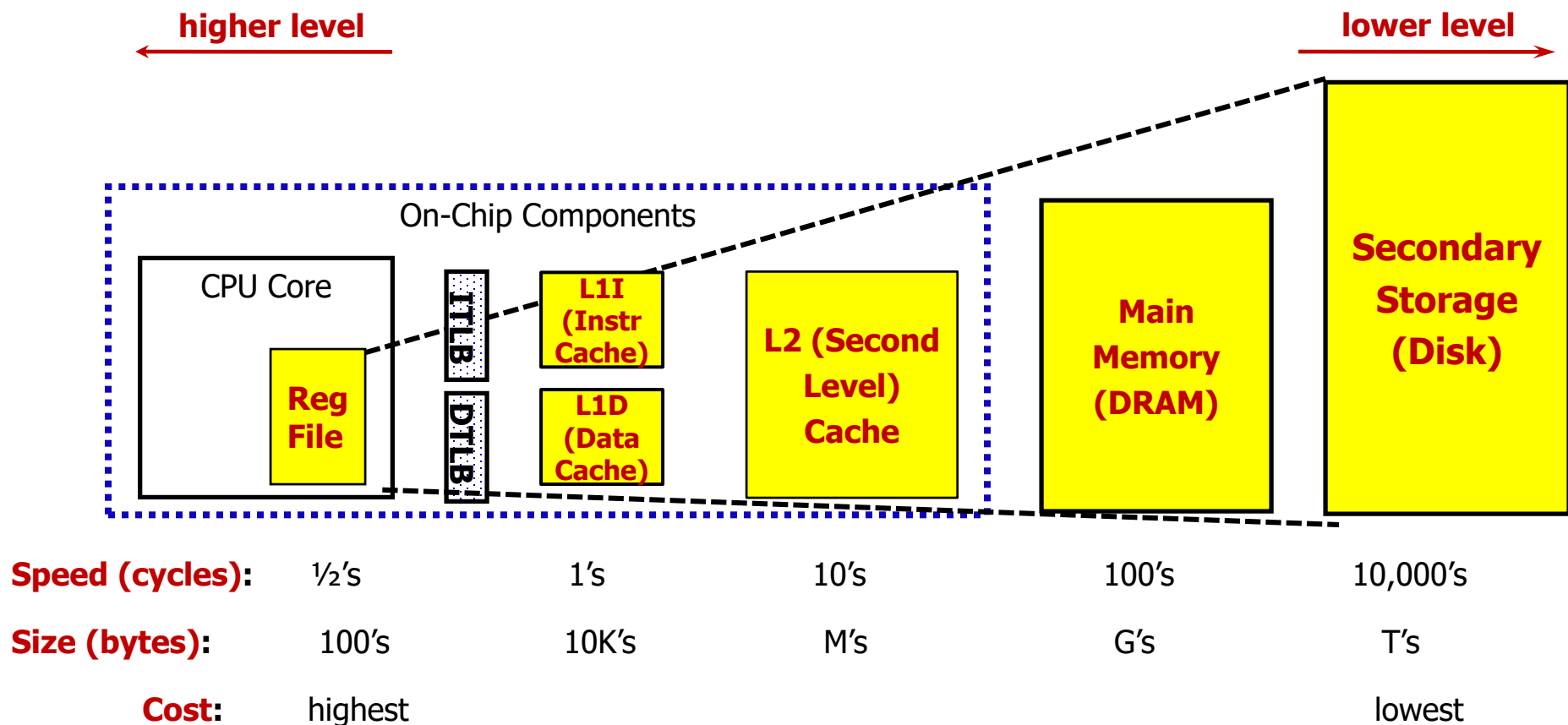
```
int A[100], B[100], C[100], D;
for (i=0; i<100; i++) {
    C[i] = A[i] * B[i] + D;
}
```

Cache



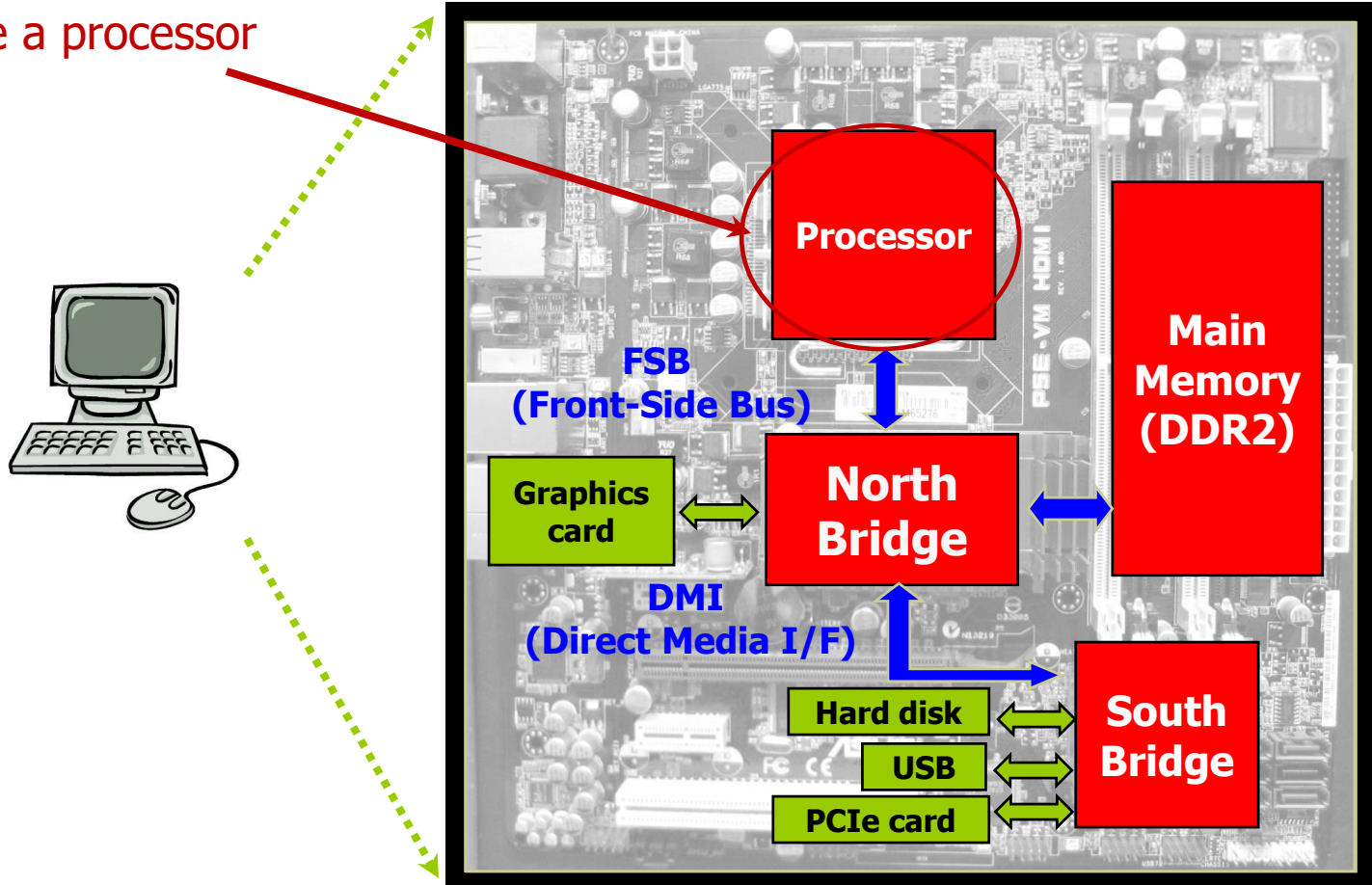
A Typical Memory Hierarchy

- Take advantage of the **principle of locality** to present the user with as much memory as is available in the **cheapest** technology at the speed offered by the **fastest** technology



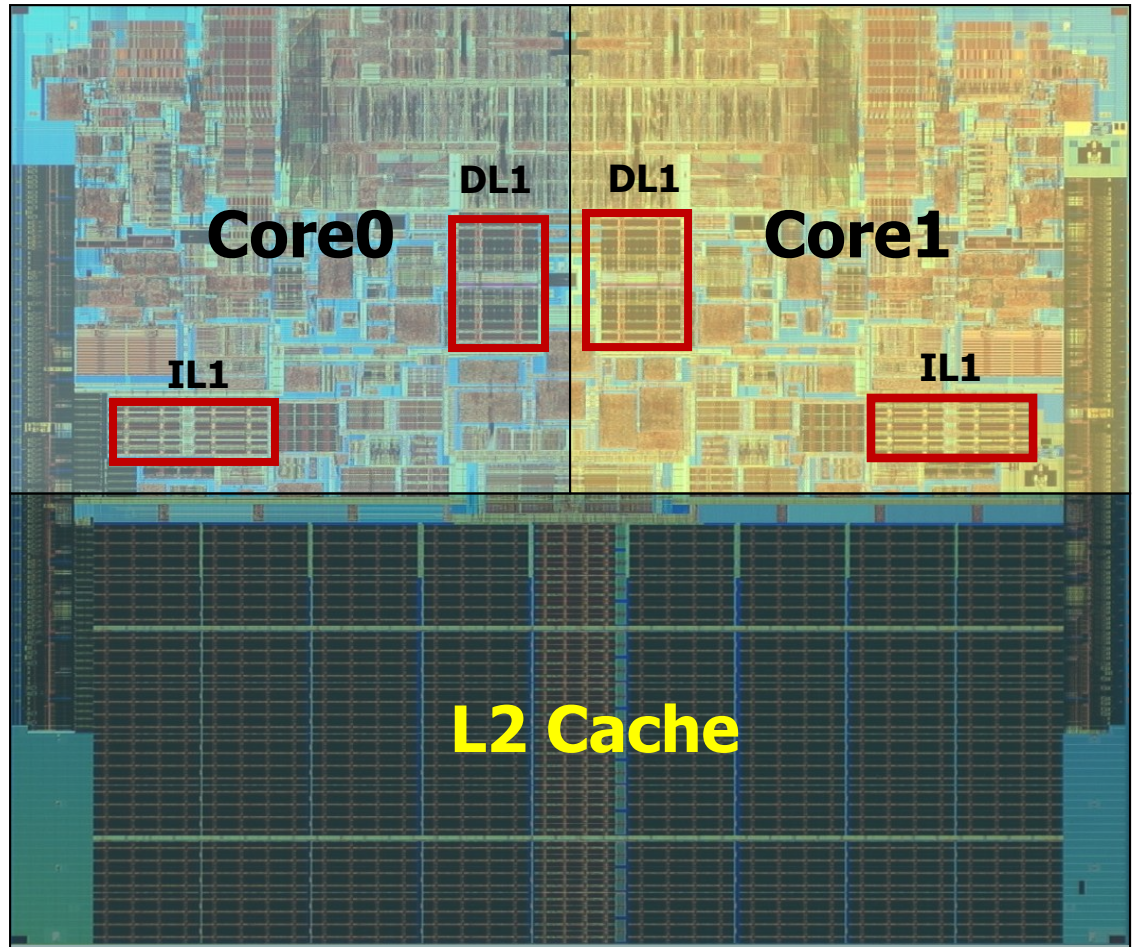
A Computer System

Caches are located inside a processor



Core 2 Duo (Intel)

L1	32 KB , 8-Way, 64 Byte/Line, LRU, WB 3 Cycle Latency
L2	4.0 MB , 16-Way, 64 Byte/Line, LRU, WB 14 Cycle Latency



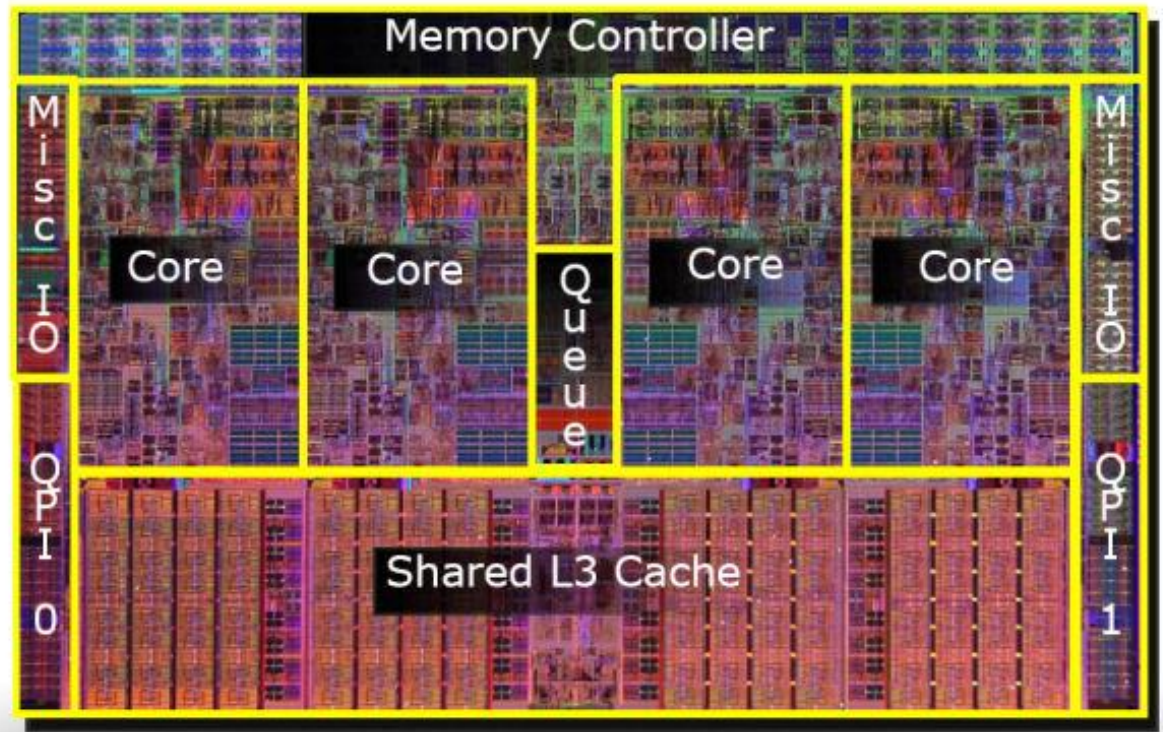
Source: <http://www.sandpile.org>

Core i7 (Intel)

- 4 cores on one chip
- Three levels of caches (L1, L2, L3) on chip
 - **L1: 32KB**, 8-way
 - **L2: 256KB**, 8-way
 - **L3: 8MB**, 16-way
- 731 million transistors in 263 mm² with 45nm technology

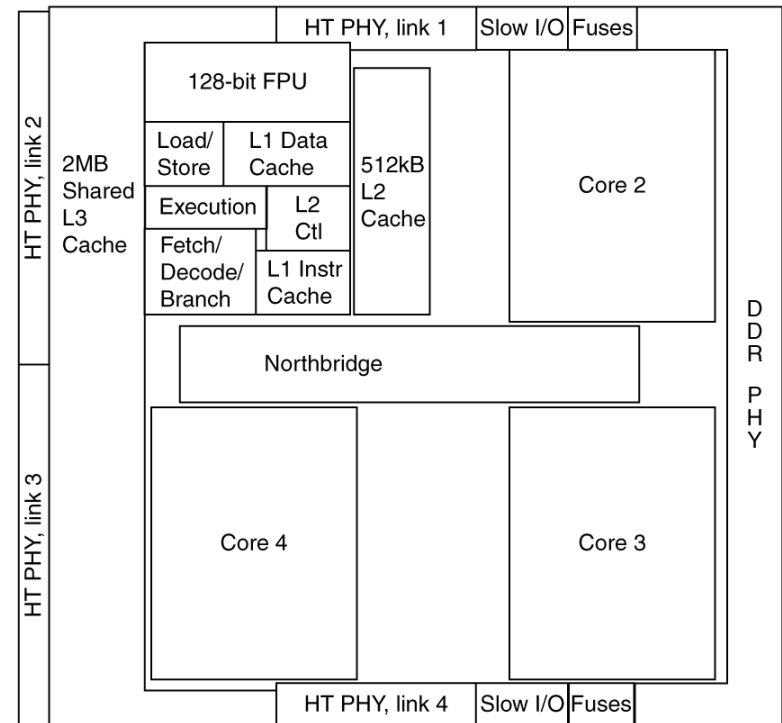
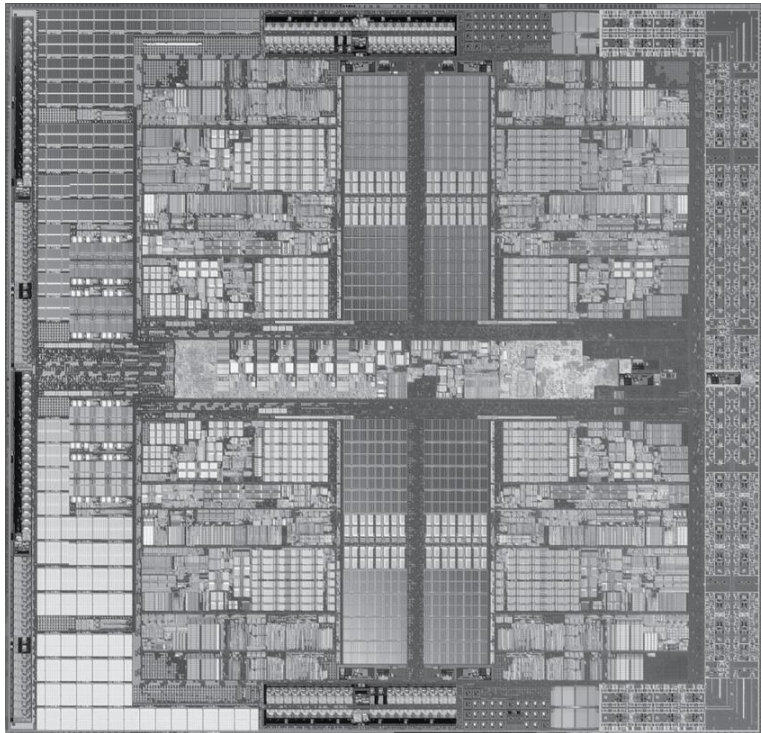


The First Nehalem Processor



Opteron (AMD) - Barcelona

- 4 cores on one chip
- Three levels of caches (L1, L2, L3) on chip
 - **L1: 64KB, L2: 512KB, L3: 2MB**
- Integrated North Bridge

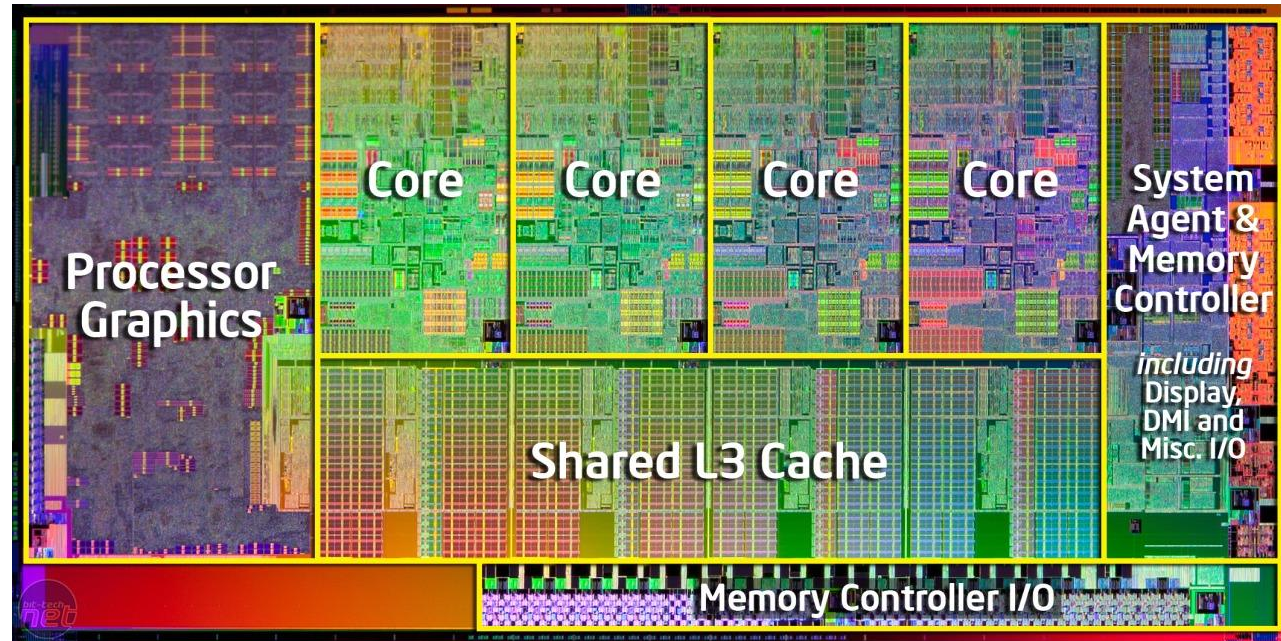


Core i7 (2nd Gen.)

2nd Generation Core i7

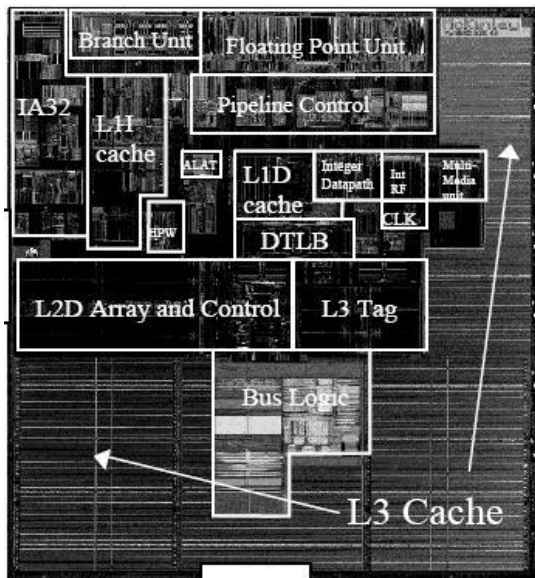
Sandy Bridge

L1	32 KB
L2	256 KB
L3	8MB



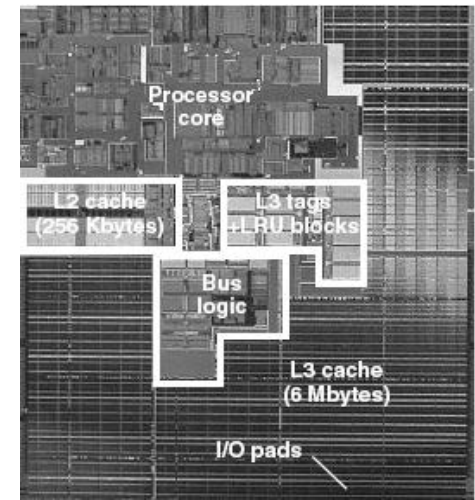
**995 million transistors
in 216 mm² with 32nm
technology**

Intel Itanium 2 (2002~)



3MB
Version
180nm
421 mm²

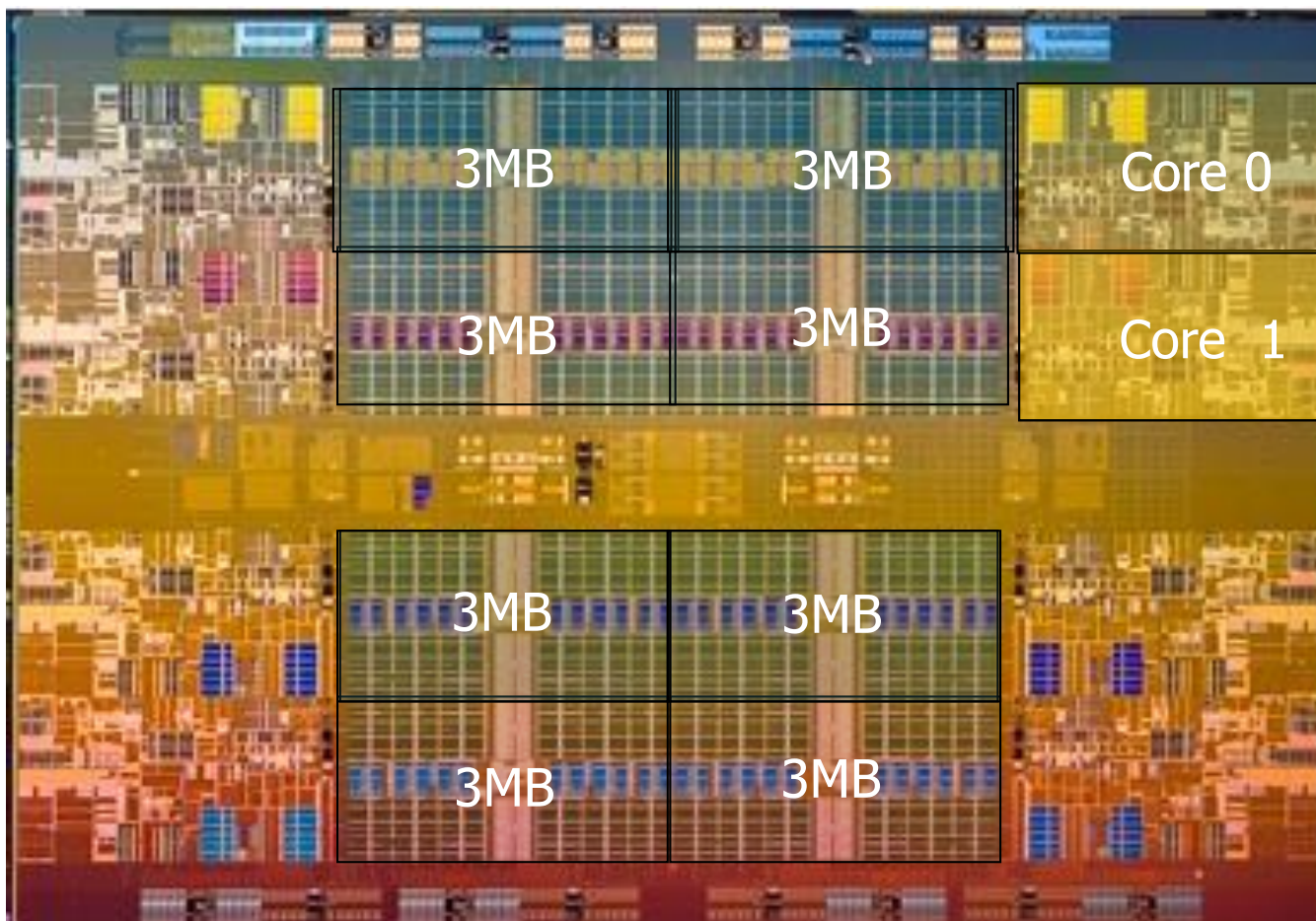
6MB
Version
130nm
374 mm²



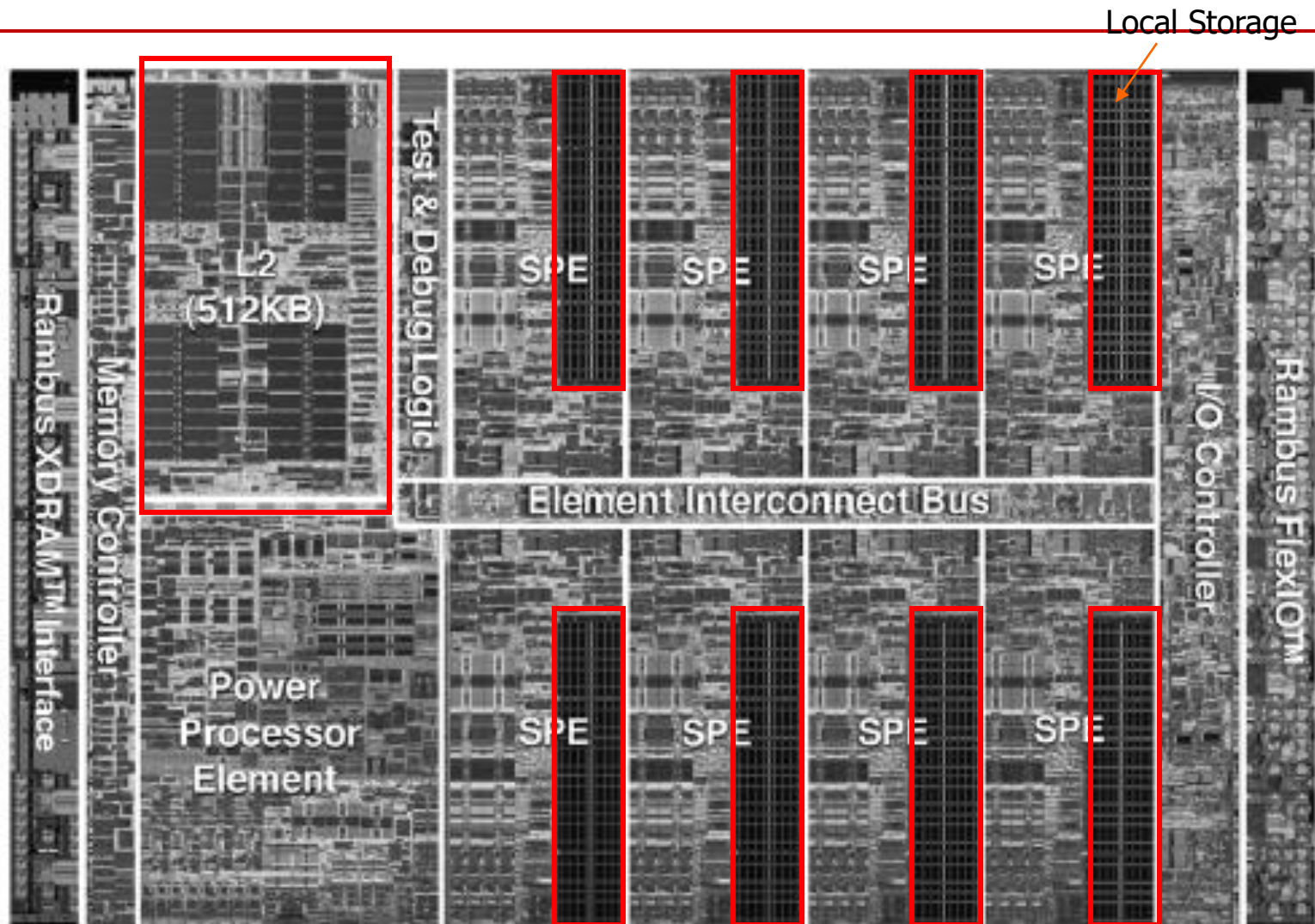
Attribute	L1 instruction cache	L1 data cache	L2	L3
Size	16 Kbytes	16 Kbytes	256 Kbytes	6 Mbytes
Line size (bytes)	64	64	128	128
No. of ways	4	4	8	24
Replacement policy	Least recently used	Not recently used	Not recently used	Not recently used
Latency (no. of cycles)	1	Integer: 1 Floating-point: NA	Integer: 5 Floating-point: 6	14
Write policy	NA	Write through	Write back	Write back
Bandwidth (Gbytes/s)				
Read	48	24	48	48
Write	NA	24	48	48

Xeon Nehalem-EX (2010)

- 24MB Shared L3

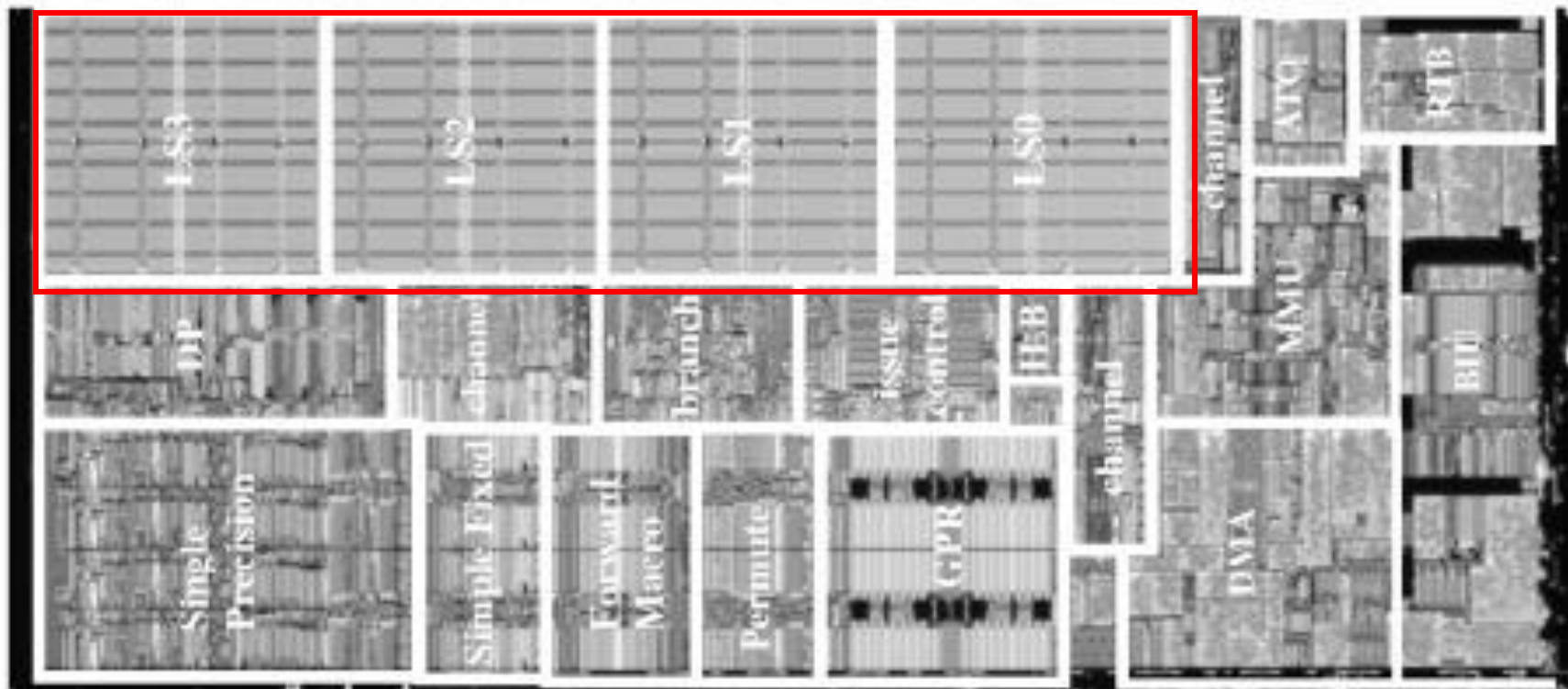


Example : STI Cell Processor



SPE = 21M transistors (14M array; 7M logic)

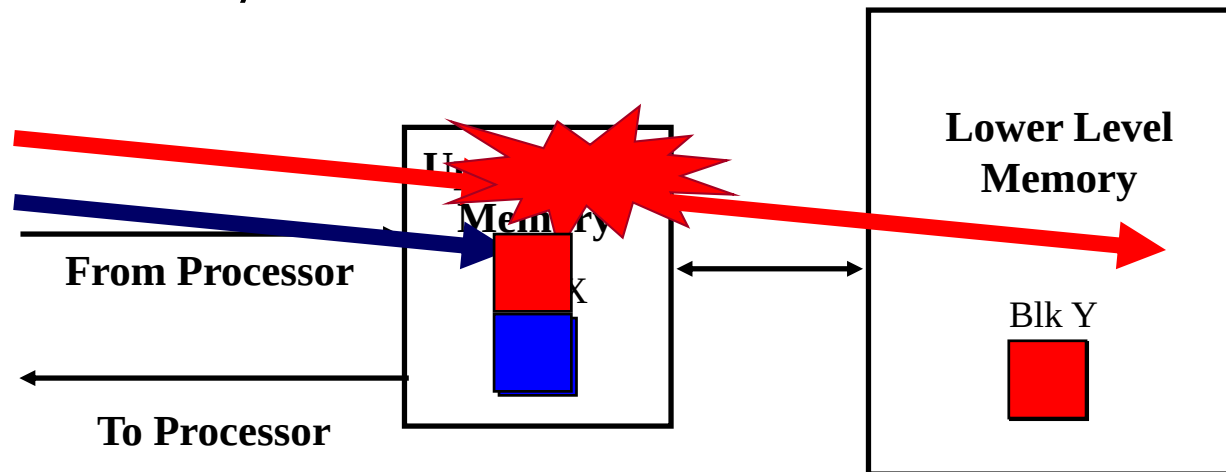
Cell Synergistic Processing Element



Each SPE contains 128 x128 bit registers,
256KB, 1-port, ECC-protected local SRAM (Not cache)

Cache Terminology

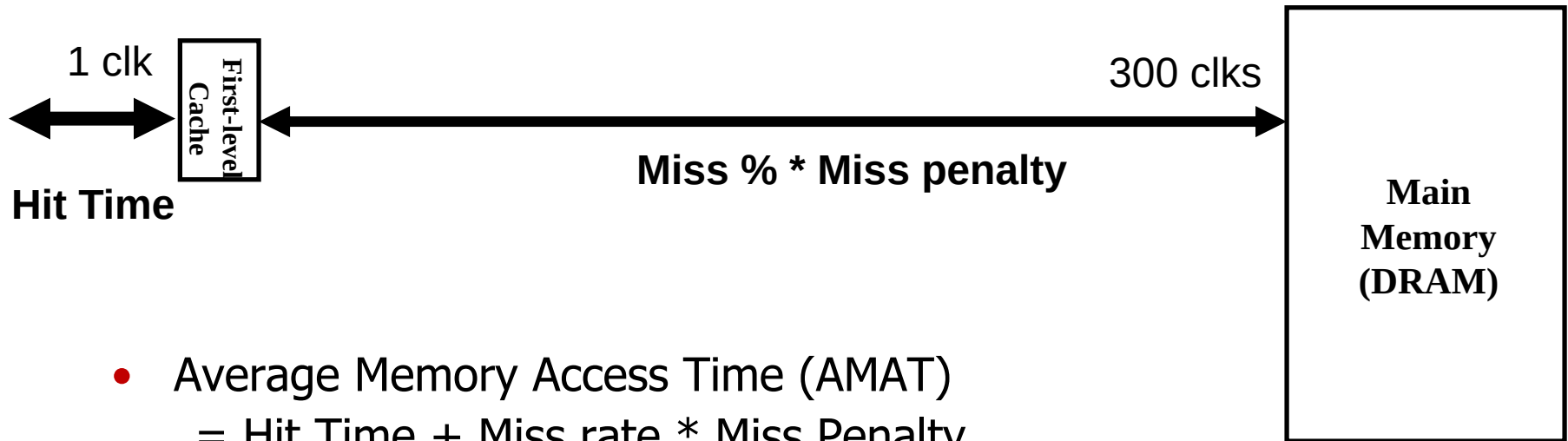
- **Hit**: data appears in some block
 - **Hit Rate**: the fraction of memory accesses found in the level
 - **Hit Time**: Time to access the level (consists of cache access time + time to determine hit)
- **Miss**: data needs to be retrieved from a block in the lower level (e.g., Block Y)
 - **Miss Rate** = $1 - (\text{Hit Rate})$
 - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block to the processor
- Hit Time $\ll$ Miss Penalty



Average Memory Access Time

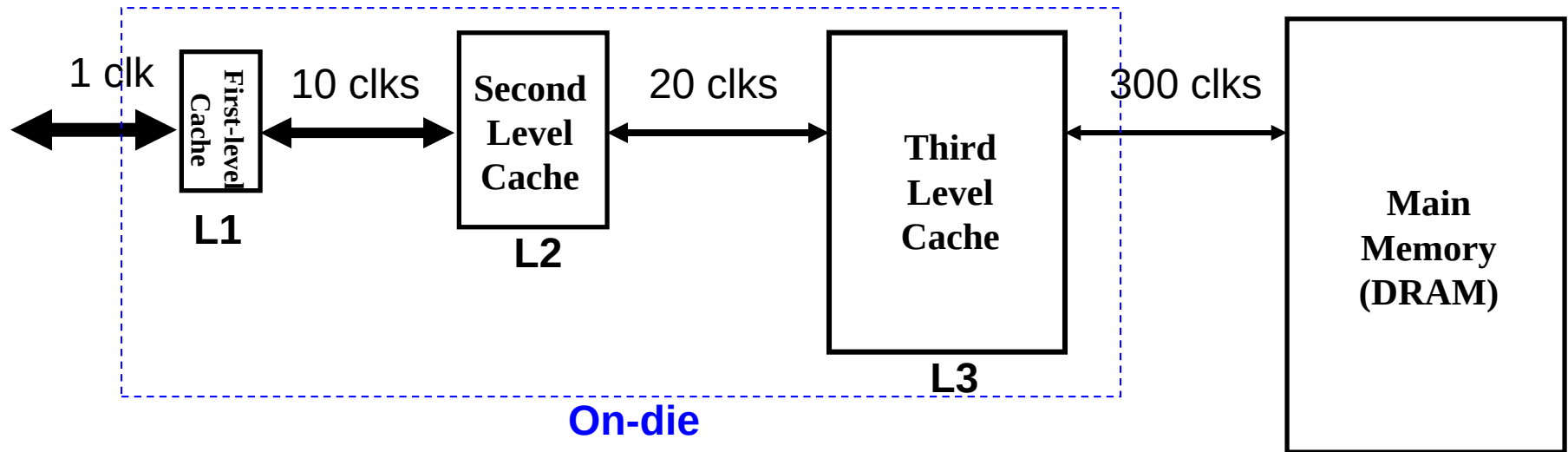
- Average memory-access time
= Hit time + Miss rate x Miss penalty
- Miss penalty: time to fetch a block from lower memory level
 - access time: function of latency
 - transfer time: function of bandwidth b/w levels
 - Transfer “one block (one cache line)” at a time
 - Transfer at the size of the memory-bus width

Memory Hierarchy Performance



- Average Memory Access Time (AMAT)
= Hit Time + Miss rate * Miss Penalty
= $T_{\text{hit}}(\text{L1}) + \text{Miss\%}(\text{L1}) * T(\text{memory})$
- Example:
 - Cache Hit = 1 cycle
 - Miss rate = 10% = 0.1
 - Miss penalty = 300 cycles
 - $\text{AMAT} = 1 + 0.1 * 300 = 31$ cycles
- Can we improve it?

Reducing Penalty: Multi-Level Cache



Average Memory Access Time (AMAT) =

$$T_{\text{hit}}(\text{L1}) + \text{Miss}\%(\text{L1}) * (T_{\text{hit}}(\text{L2}) + \text{Miss}\%(\text{L2}) * (T_{\text{hit}}(\text{L3}) + \text{Miss}\%(\text{L3}) * T(\text{memory})))$$

AMAT of multi-level memory

$$= T_{\text{hit}}(\text{L1}) + \text{Miss}\%(\text{L1}) * T_{\text{miss}}(\text{L1})$$

$$= T_{\text{hit}}(\text{L1}) + \text{Miss}\%(\text{L1}) * \{ T_{\text{hit}}(\text{L2}) + \text{Miss}\%(\text{L2}) * T_{\text{miss}}(\text{L2}) \}$$

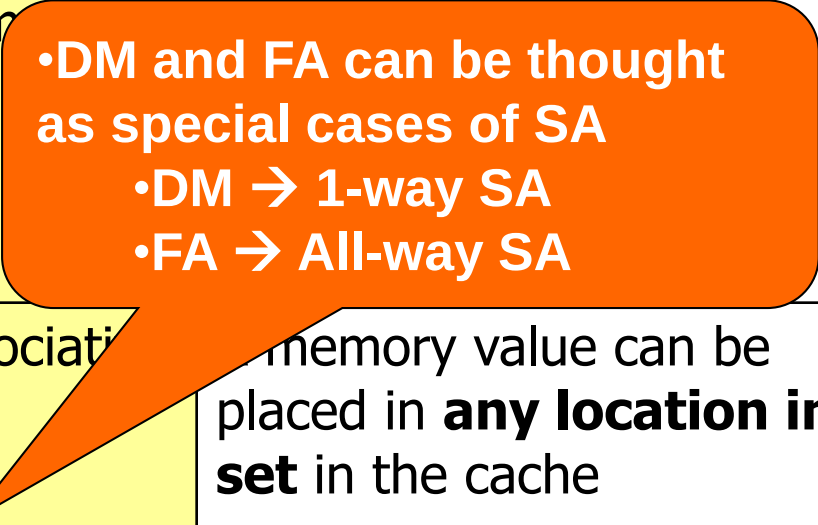
$$= T_{\text{hit}}(\text{L1}) + \text{Miss}\%(\text{L1}) * \{ T_{\text{hit}}(\text{L2}) + \text{Miss}\%(\text{L2}) * [T_{\text{hit}}(\text{L3}) + \text{Miss}\%(\text{L3}) * T(\text{memory})] \}$$

AMAT Example

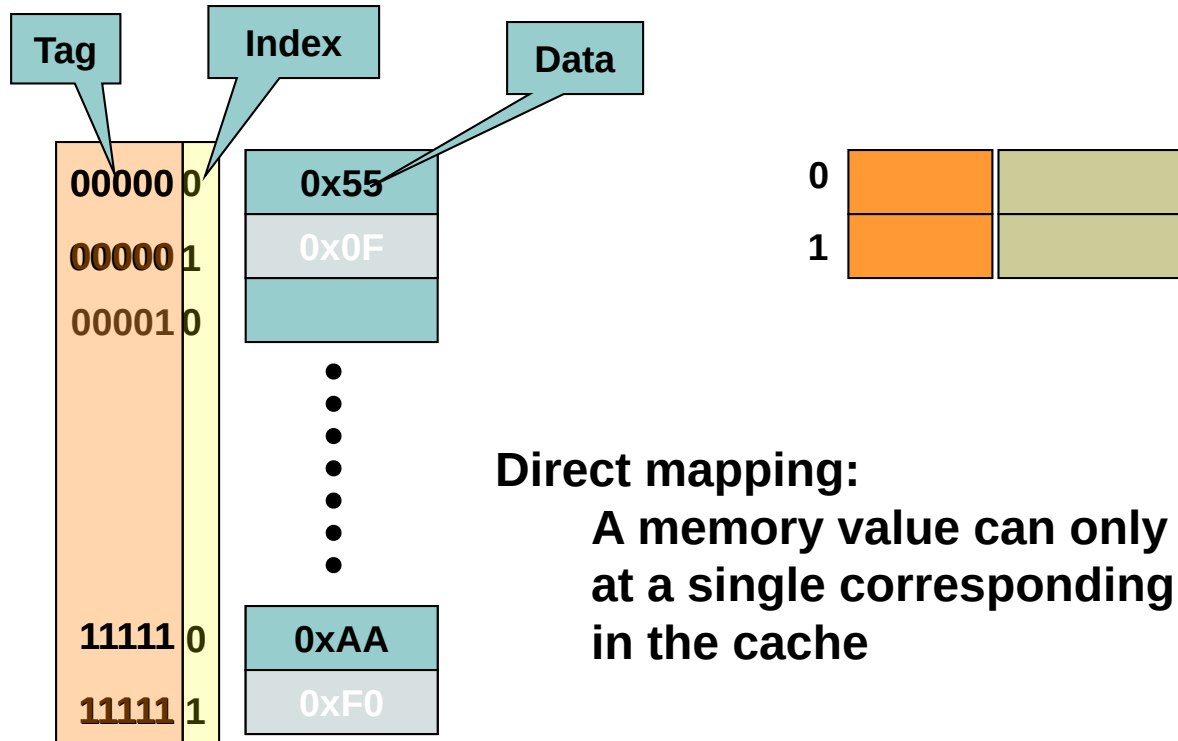
$$AMAT = T_{hit}(L1) + Miss\%(L1) * (T_{hit}(L2) + Miss\%(L2) * (T_{hit}(L3) + Miss\%(L3) * T(memory)))$$

- Example:
 - Miss rate L1=10%, $T_{hit}(L1) = 1$ cycle
 - Miss rate L2=5%, $T_{hit}(L2) = 10$ cycles
 - Miss rate L3=1%, $T_{hit}(L3) = 20$ cycles
 - $T(memory) = 300$ cycles
- $AMAT = ?$
 - 2.115 (compare to 31 with no multi-levels)
14.7x speed-up!

Types of Caches

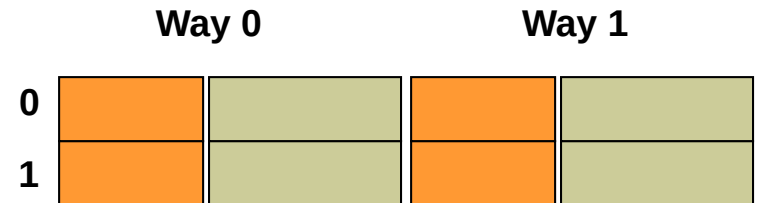
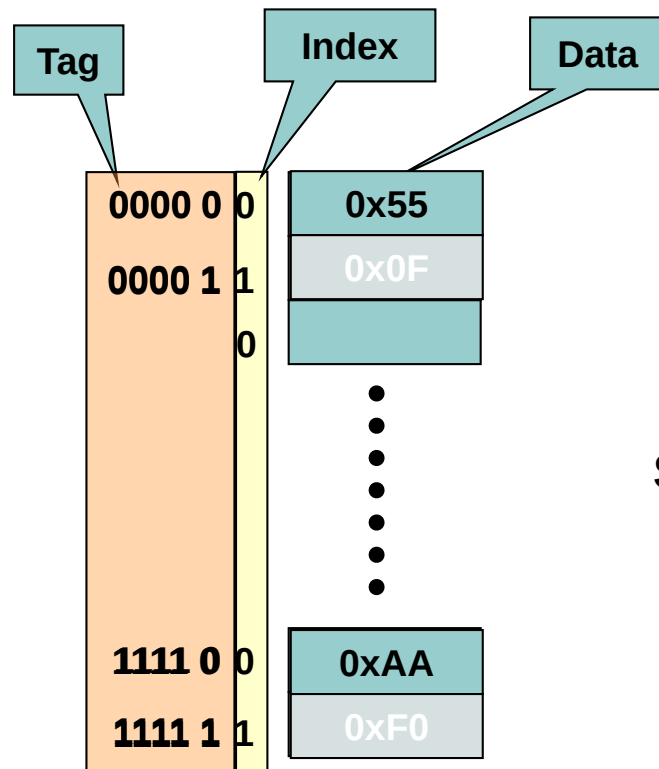
Type of cache	Mapping of data from memory to cache	Complexity of searching the cache
Direct mapping (DM)	 •DM and FA can be thought as special cases of SA •DM → 1-way SA •FA → All-way SA A memory value can be placed in any location in a set in the cache	Fast indexing mechanism
Set-associative (SA)	A memory value can be placed in any location in the cache	Slightly more involved search mechanism
Fully-associative (FA)	A memory value can be placed in any location in the cache	Extensive hardware resources required to search (CAM)

Direct Mapping



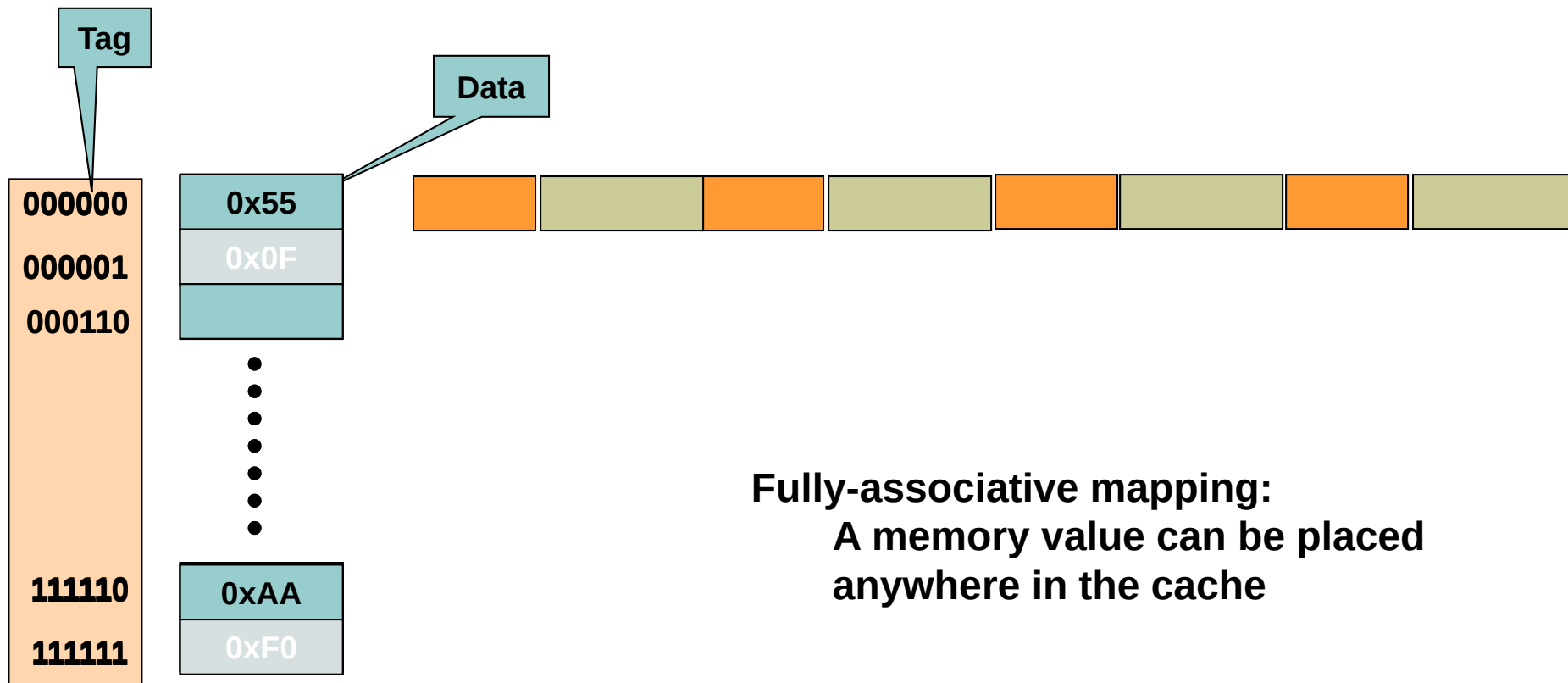
Direct mapping:
A memory value can only be placed
at a single corresponding location
in the cache

Set Associative Mapping (2-Way)

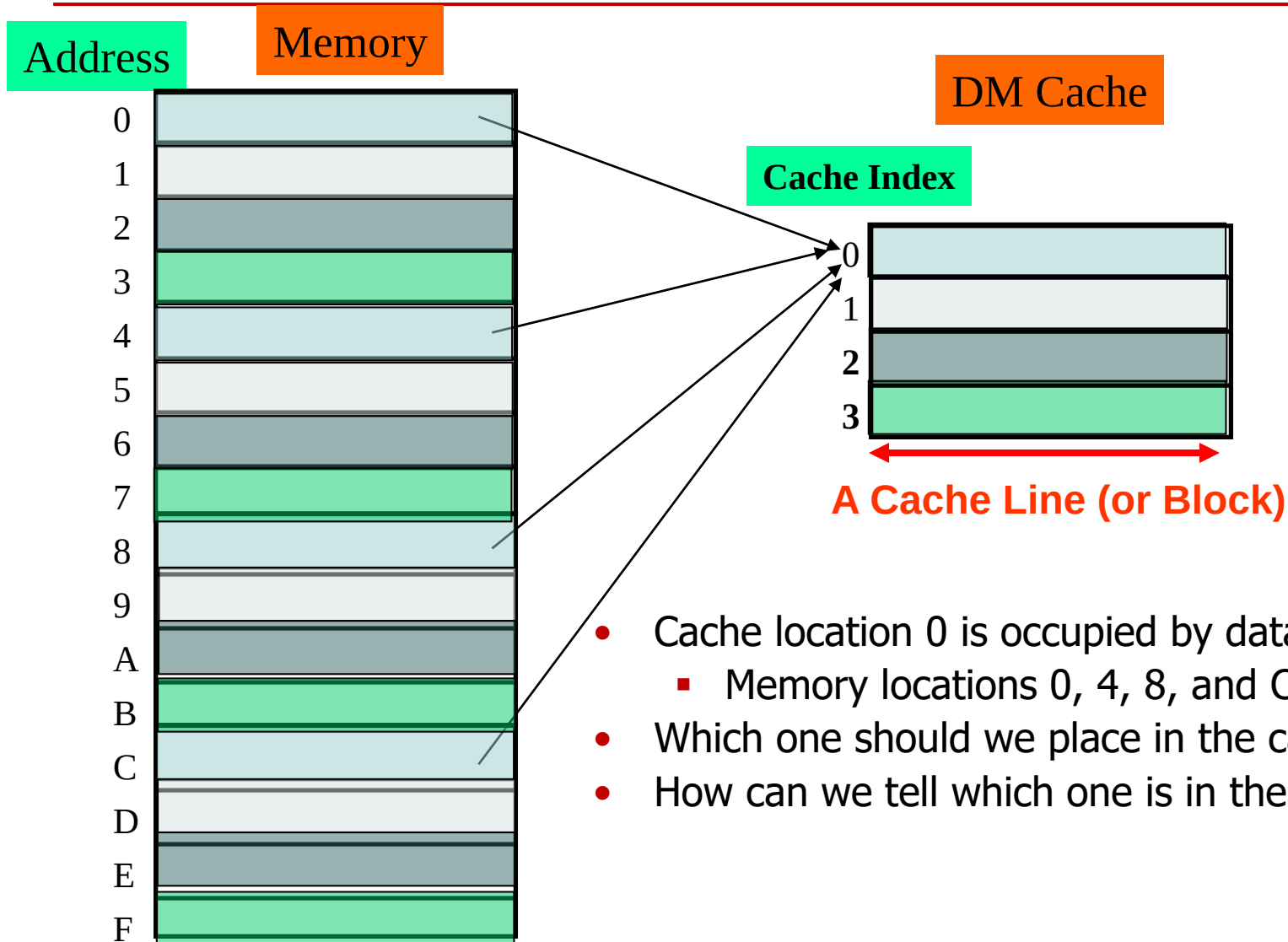


Set-associative mapping:
A memory value can be placed in any location of a set in the cache

Fully Associative Mapping



Direct Mapped Cache

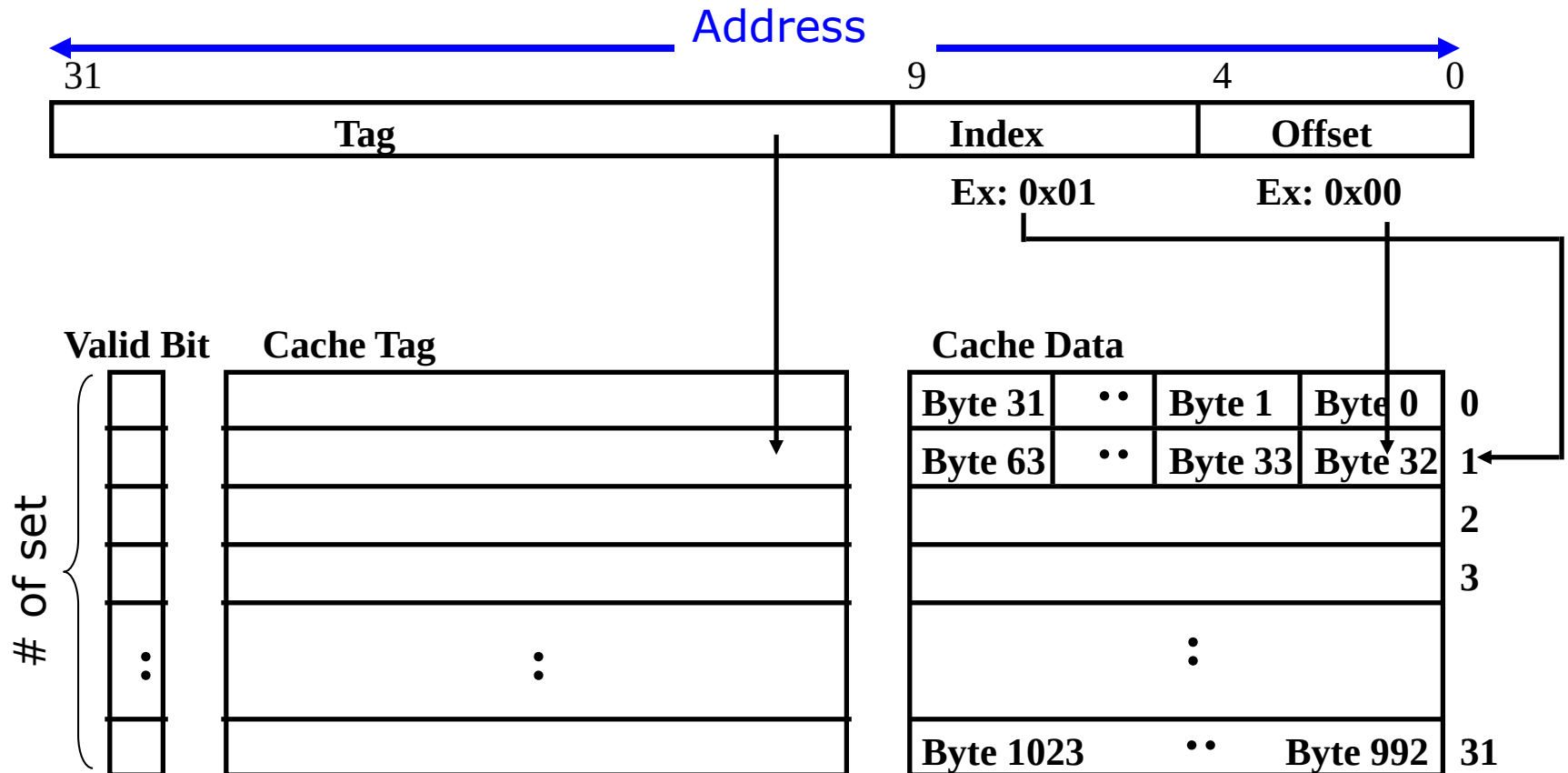


Three (or Four) Cs (Cache Miss Terms)

- Compulsory Misses:
 - cold start misses (Caches do not have valid data at the start of the program)
- Capacity Misses:
 - Increase cache size
- Conflict Misses:
 - Increase cache size and/or associativity.
 - Associative caches reduce conflict misses
- Coherence Misses:
 - In multiprocessor systems (later lectures...)

Example: 1KB DM Cache, 32-byte Lines

- The lowest M bits are the Offset (Line Size = 2^M)
- Index = $\log_2$ (# of sets)



Example of Caches

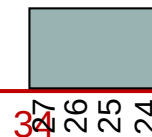
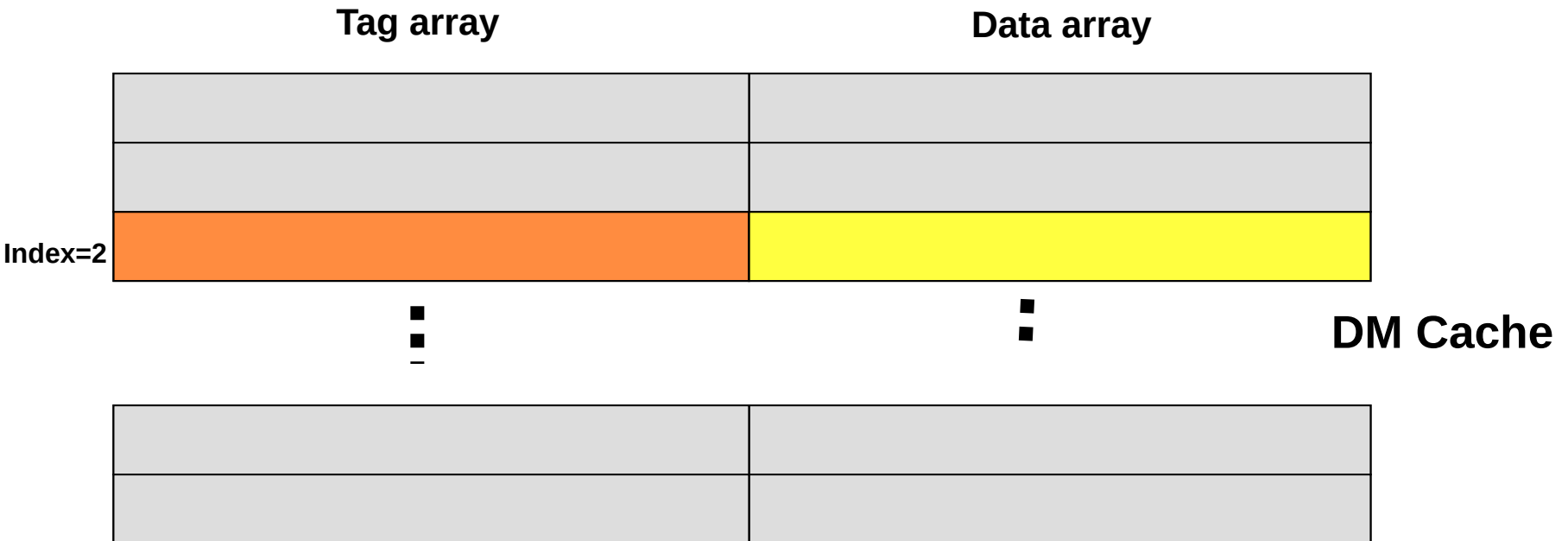
- Given a 2MB, direct-mapped physical caches, line size=64bytes
 - Support up to 52-bit physical address
 - Tag size?
-
- Now change it to 16-way, Tag size?
-
- How about if it's fully associative, Tag size?

Example: 1KB DM Cache, 32-byte Lines

- $\perp_W$ from 0x77FF1C68

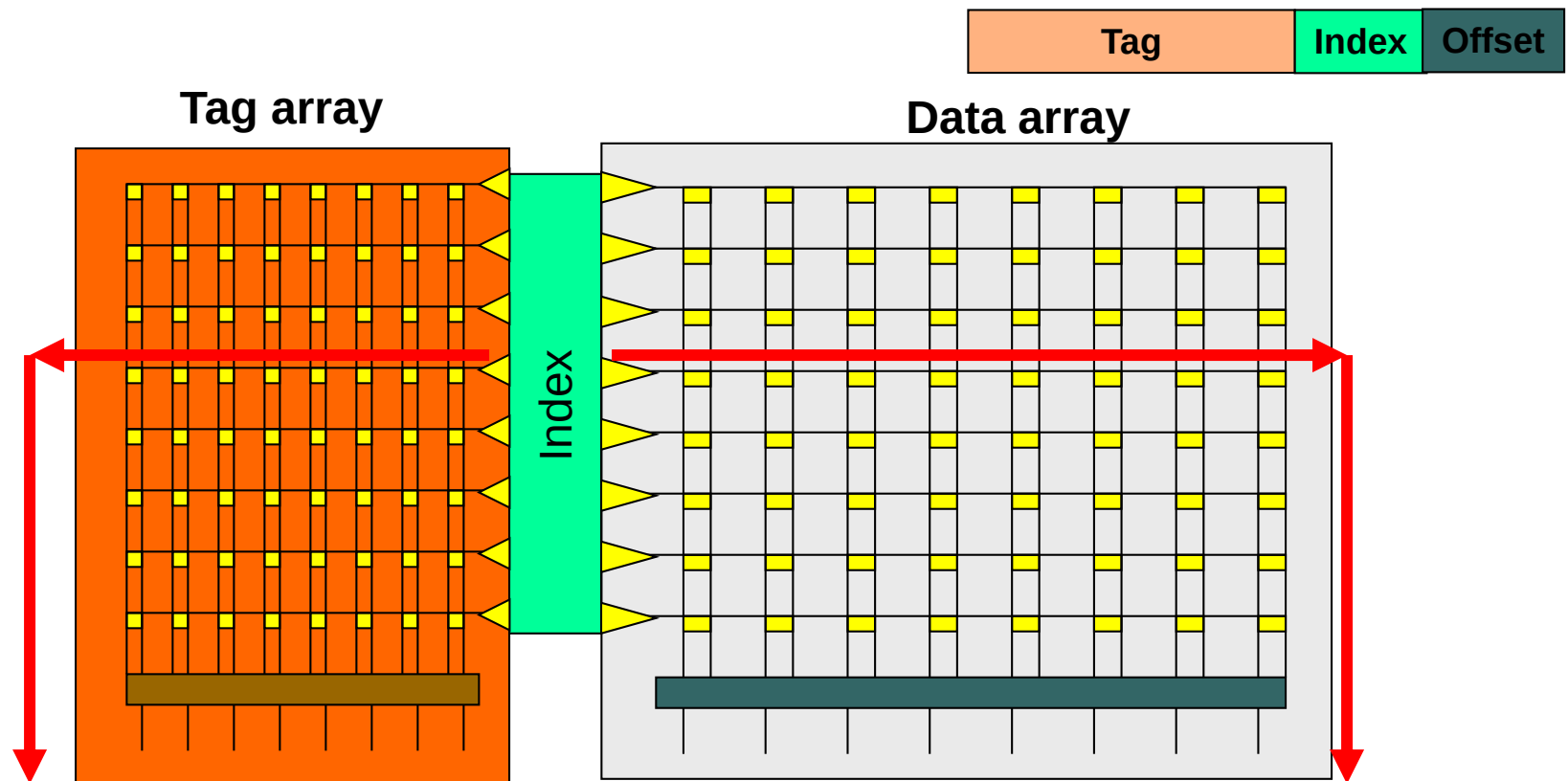


77FF1C68 = 0111 0111 1111 1111 0001 1100 0101 1000



DM Cache Speed Advantage

- Tag and data access happen in parallel
 - Faster cache access!



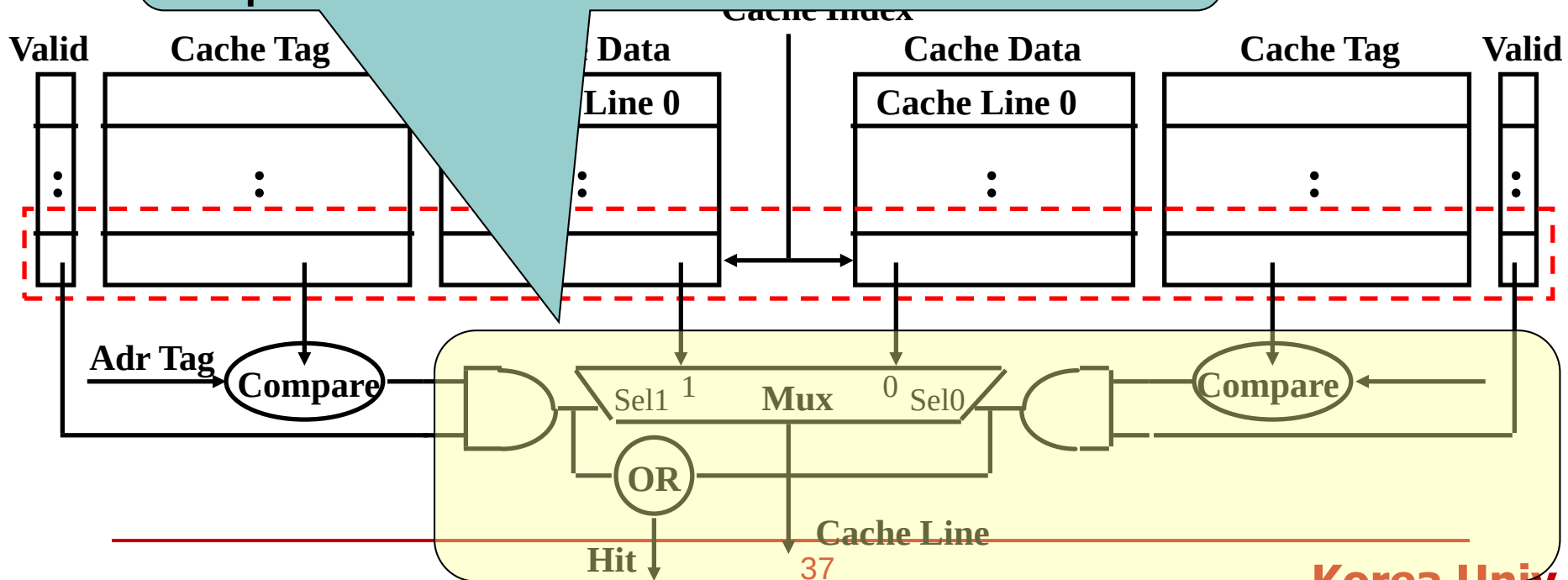
Associative Caches Reduce Conflict Misses

- Set associative (SA) cache
 - multiple possible locations in a set
- Fully associative (FA) cache
 - any location in the cache
- Hardware and speed overhead
 - Comparators
 - Multiplexors
 - Data selection only after Hit/Miss determination (i.e., after tag comparison)

Set Associative Cache (2-way)

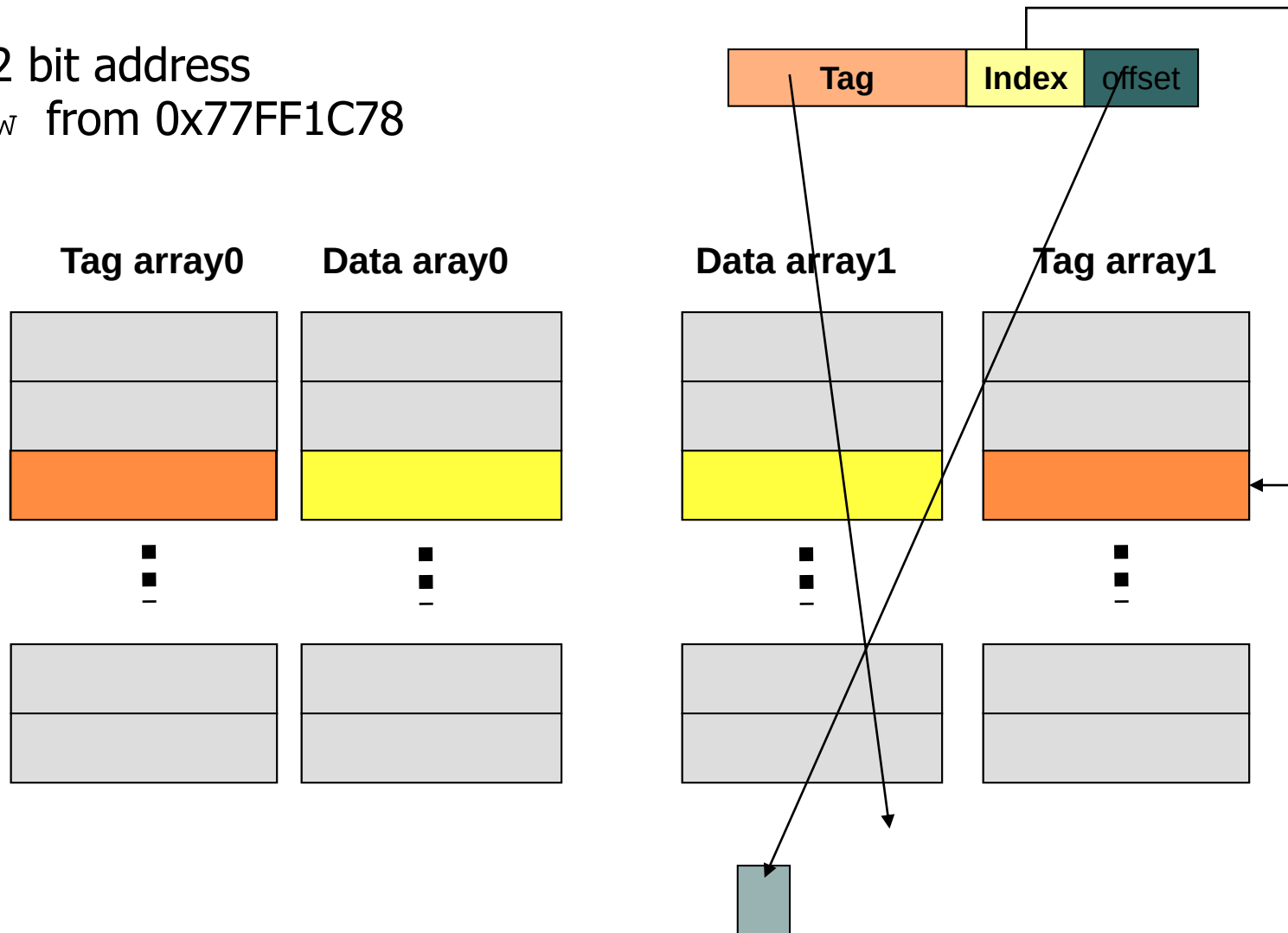
- Cache index selects a “set” from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result

• Additional circuitry as compared to DM caches
• Makes SA caches slower to access than DM of comparable size

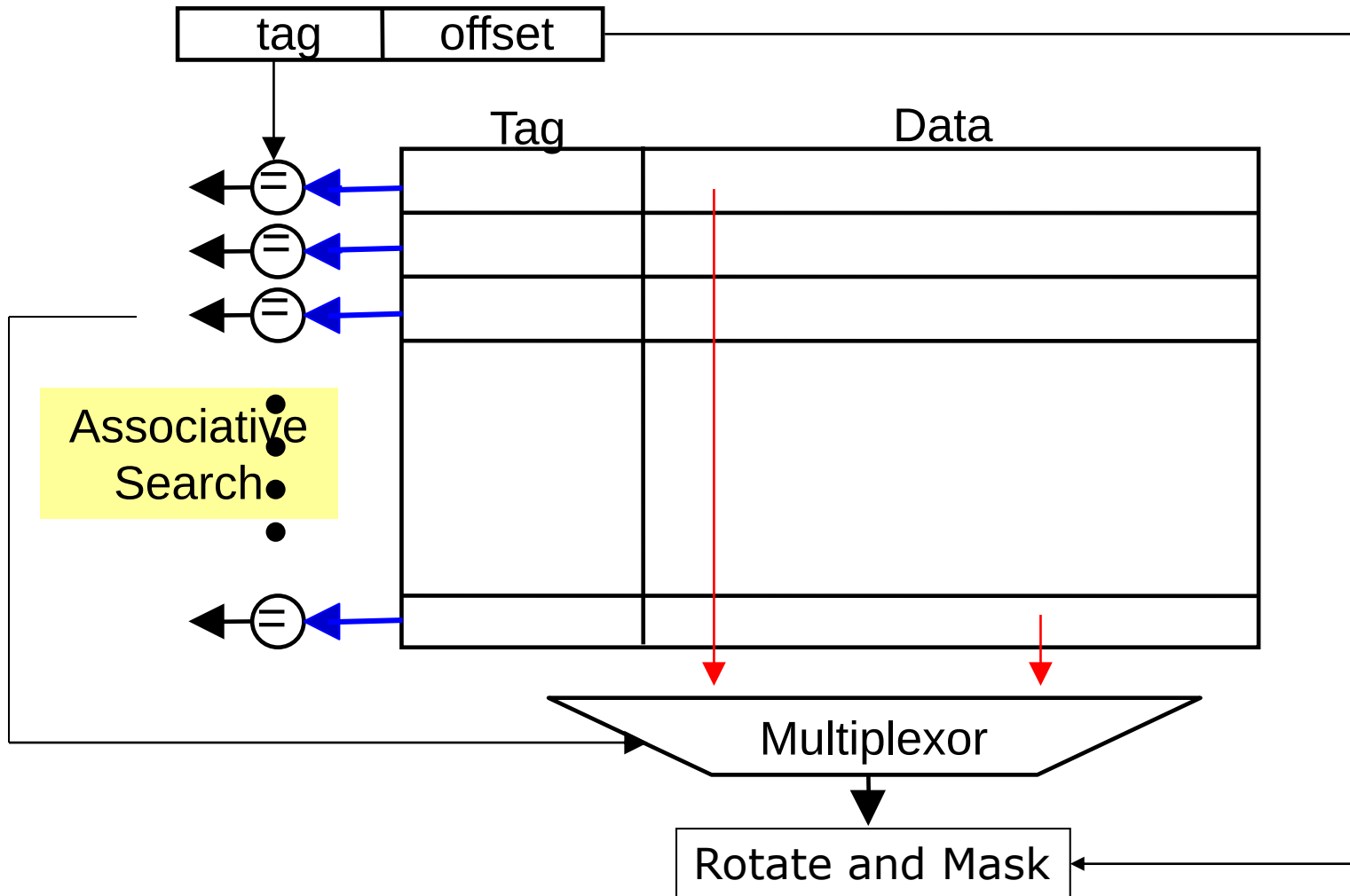


Set-Associative Cache (2-way)

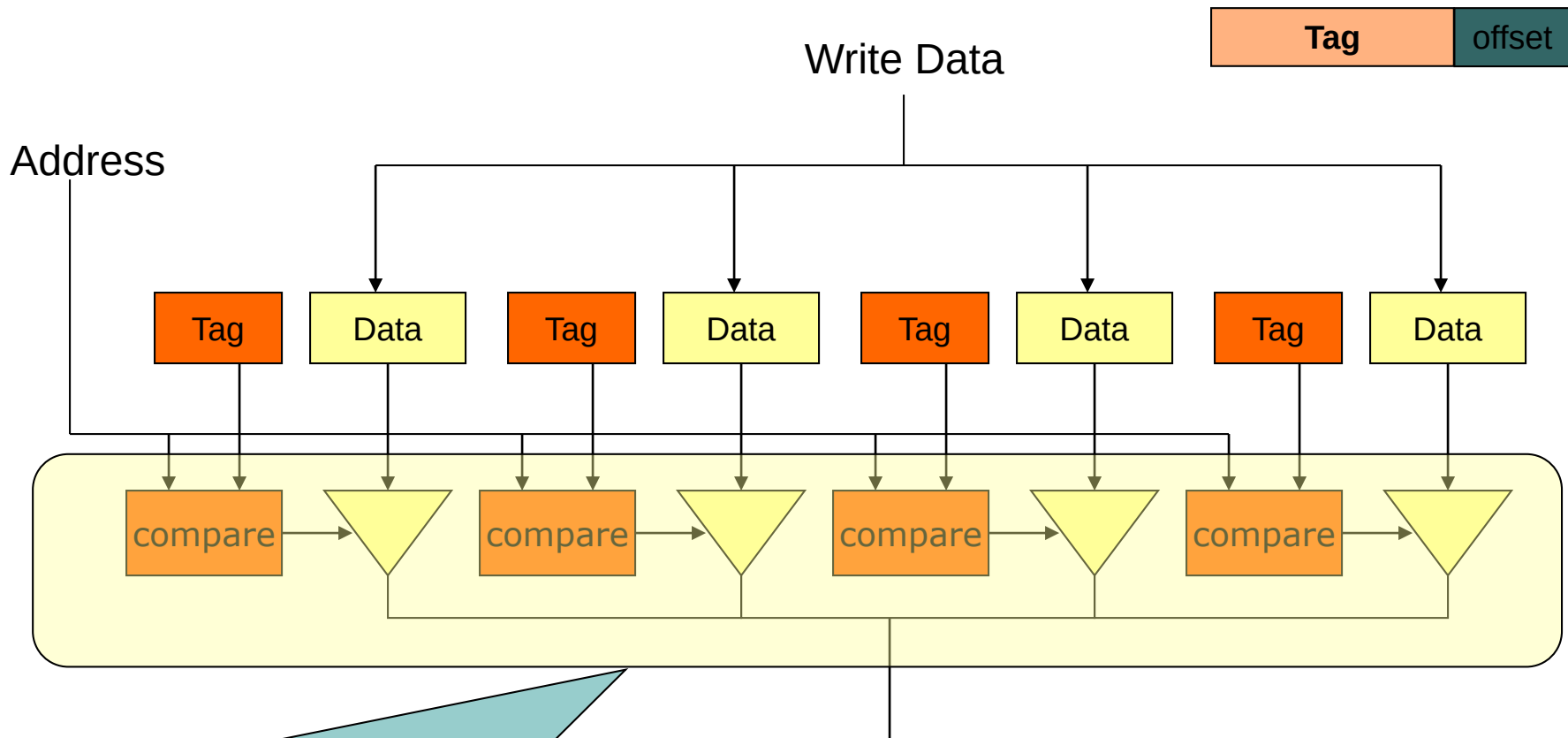
- 32 bit address
- lw from 0x77FF1C78



Fully Associative Cache



Fully Associative Cache

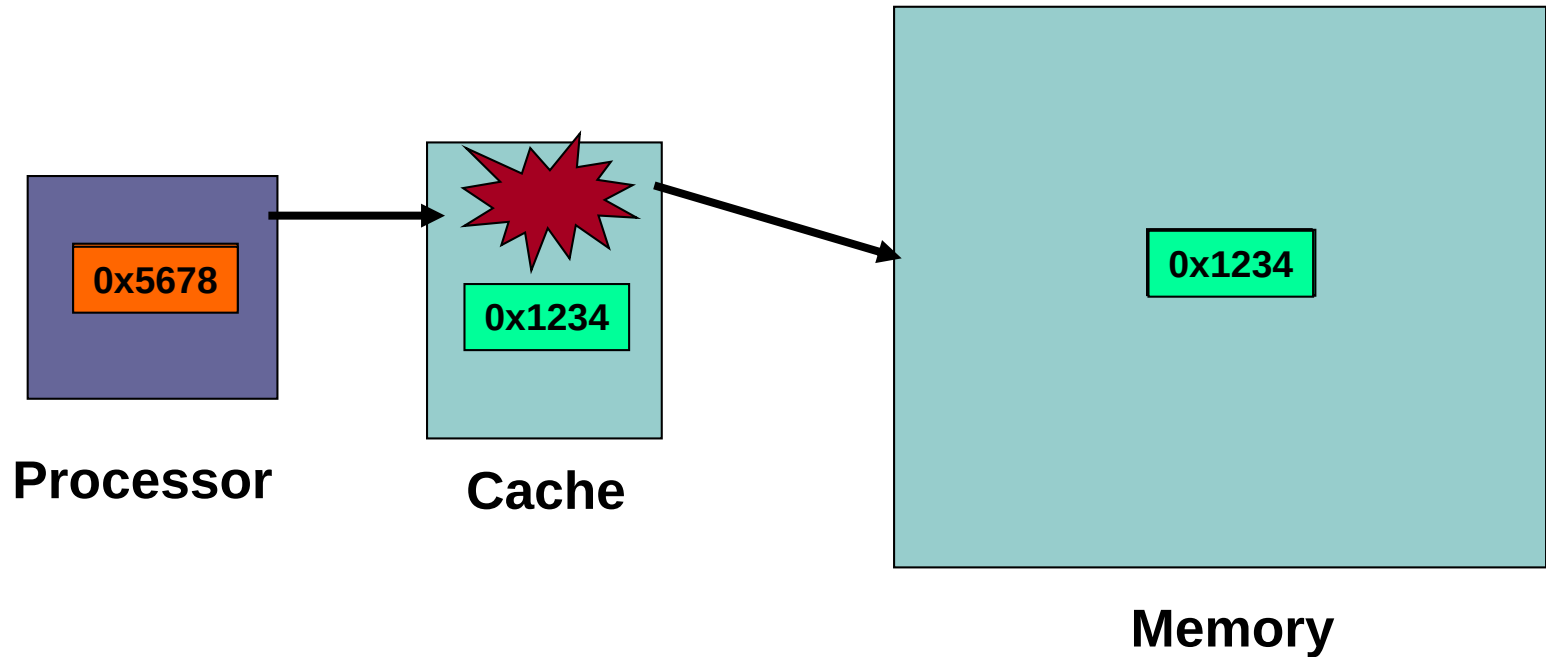


Additional circuitry as compared to DM caches
More extensive than SA caches
Makes FA caches slower to access than either DM or SA of comparable size

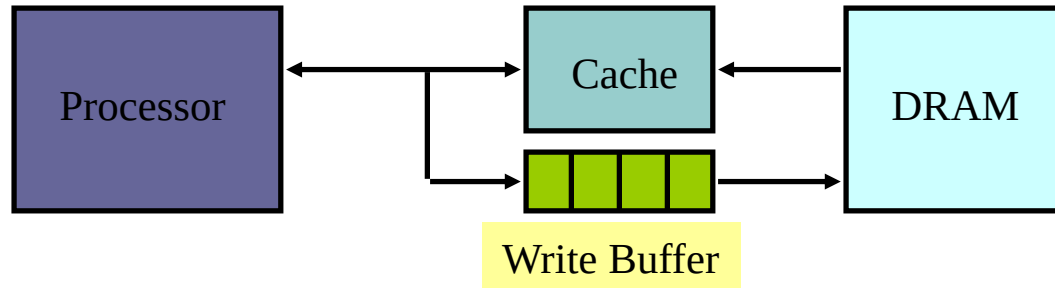
Cache Write Policy

- **Write-through** -The value is written to both the cache line and to the lower-level memory.
- **Write-back** - The value is written only to the cache line. The modified cache line is written to main memory only when it has to be replaced.
 - Is the cache line clean (holds the same value as memory) or dirty (holds a different value than memory)?

Write-through Policy

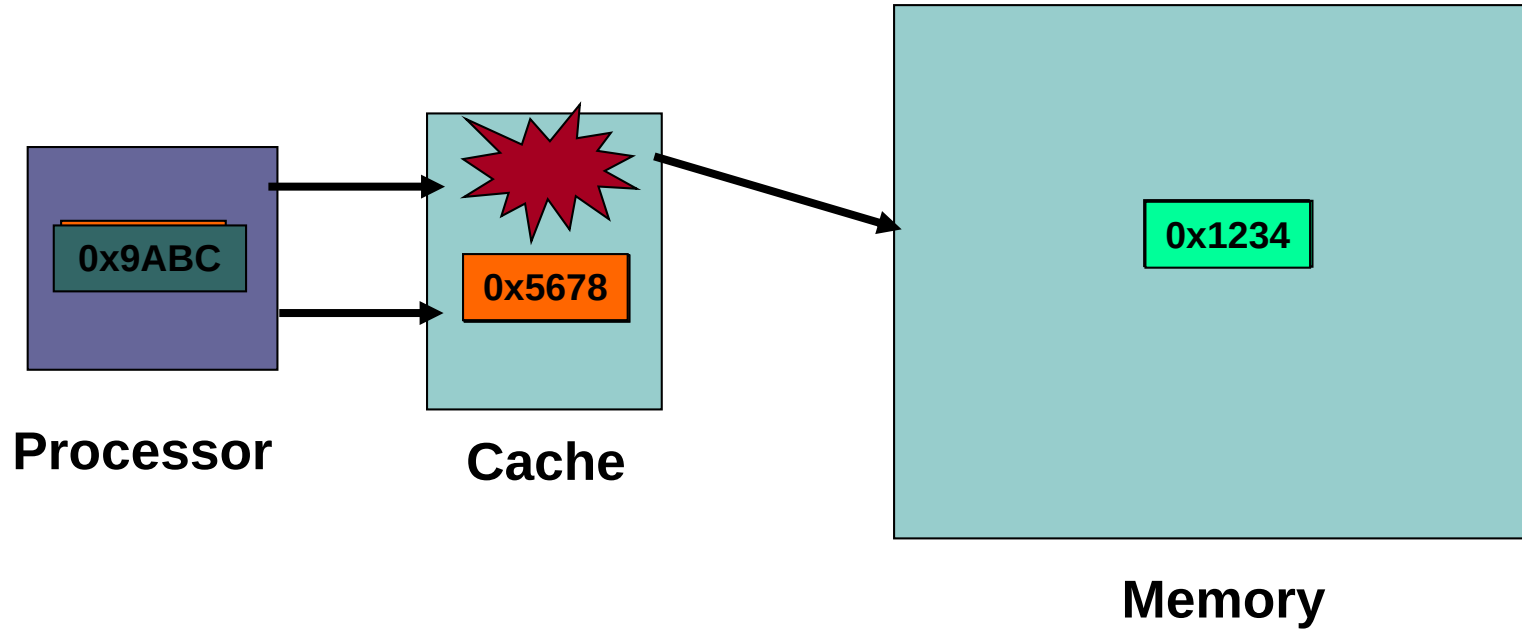


Write Buffer



- Processor: writes data into the cache and the write buffer
- Memory controller: writes contents of the buffer to memory
- Write buffer is a FIFO structure:
 - Typically 4 to 8 entries
 - Desirable: Occurrence of Writes $\ll$ DRAM write cycles
- Memory system designer's nightmare:
 - Write buffer saturation (i.e., Writes $\rightarrow$ DRAM write cycles)

Writeback Policy



On Write Miss

- **Write-allocate**

- The line is allocated on a write miss, followed by the write hit actions above.
- Write misses first act like read misses

- **No write-allocate**

- Write misses do not interfere cache
- Line is only modified in the lower level memory

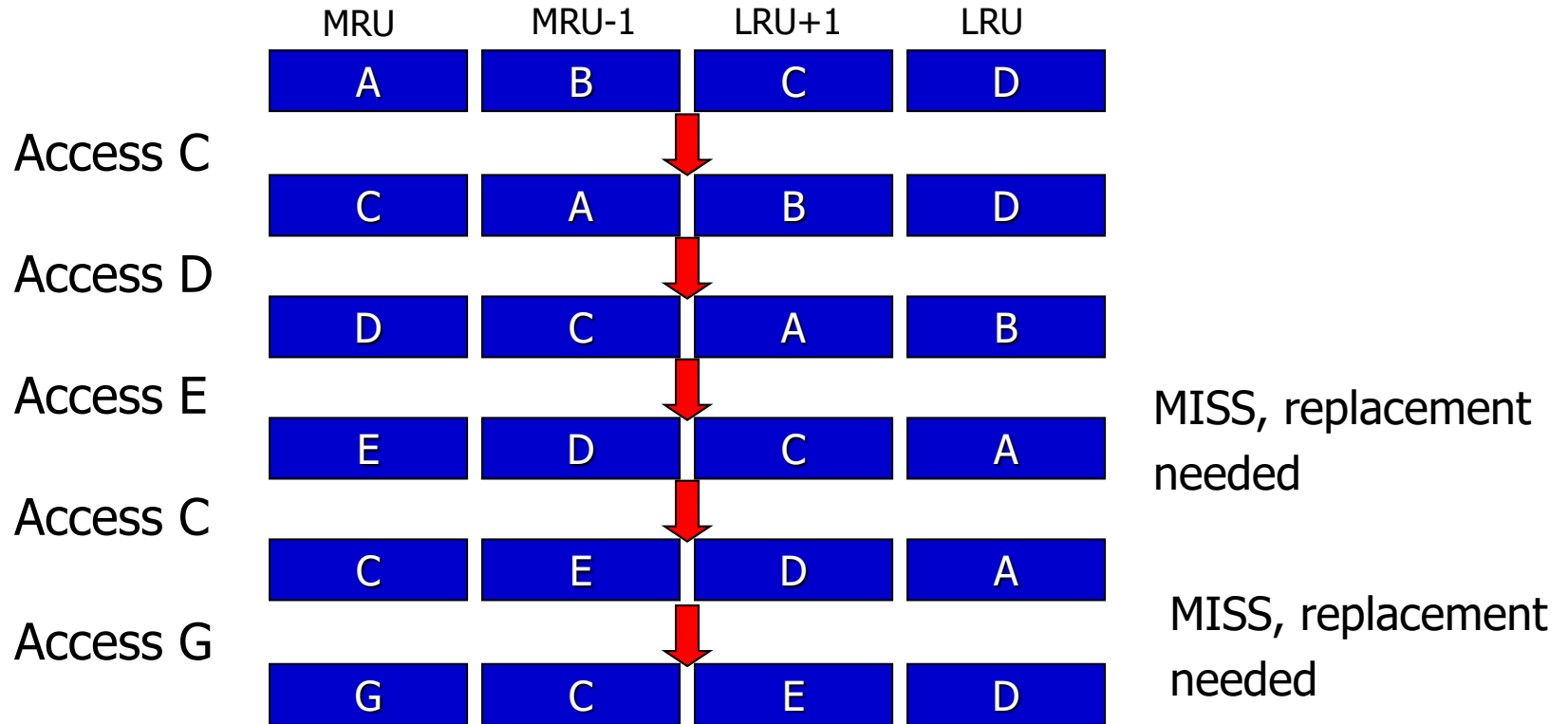
Quick recap

- Processor-memory performance gap
- Memory hierarchy exploits program locality to reduce AMAT
- Types of Caches
 - Direct mapped
 - Set associative
 - Fully associative
- Cache policies
 - Write through vs. Write back
 - Write allocate vs. No write allocate

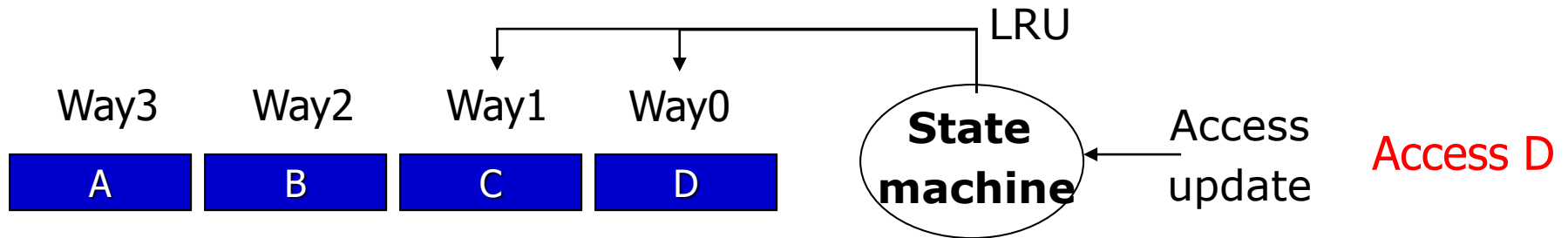
Cache Replacement Policy

- Random
 - Replace a randomly chosen line
- FIFO
 - Replace the oldest line
- LRU (Least Recently Used)
 - Replace the least recently used line
- NRU (Not Recently Used)
 - Replace one of the lines that is not recently used
 - In Itanium2 L1 Dcache, L2 and L3 caches

LRU Policy



LRU From Hardware Perspective



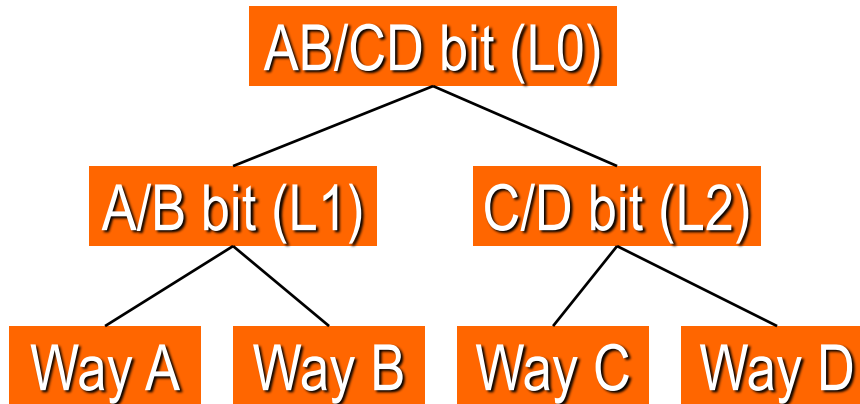
LRU policy increases cache access times

Additional hardware bits needed for LRU state machine

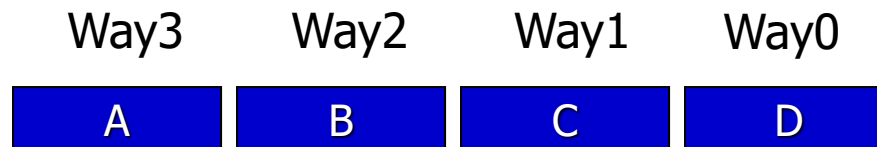
LRU Algorithms

- True LRU
 - Expensive in terms of speed and hardware
 - Need to remember the order in which all N lines were last accessed
 - $N!$ scenarios – **$O(\log N!)$** $\approx$ **$O(N \log N)$** LRU bits
 - 2-ways $\rightarrow$ AB BA = 2 = 2!
 - 3-ways $\rightarrow$ ABC ACB BAC BCA CAB CBA = 6 = 3!
- Pseudo LRU: **$O(N)$**
 - Approximates LRU policy with a binary tree

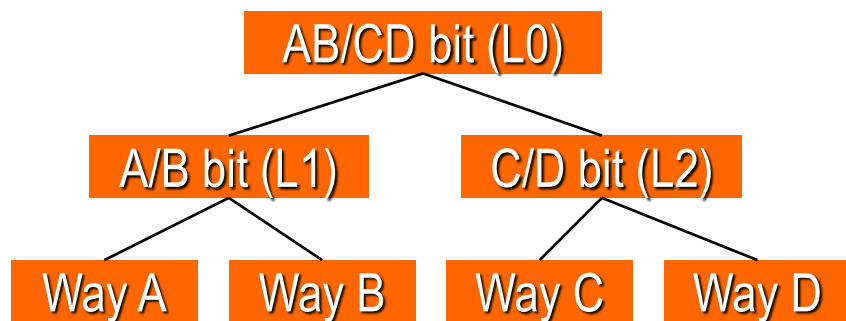
Pseudo LRU Algorithm (4-way SA)



- Tree-based
- $O(N)$: 3 bits for 4-way
- Cache ways are the leaves of the tree
- Combine ways as we proceed towards the root of the tree



Pseudo LRU Algorithm



L2L1L0 = 000,
there is a hit in Way B, what is
the new updated L2L1L0?

- Less hardware than LRU
- Faster than LRU

L2L1L0 = 001,
a way needs to be replaced,
which way would be
chosen?

LRU update algorithm

	CD	AB	AB/CD
Way hit	L2	L1	L0
Way A	---	1	1
Way B	---	0	1
Way C	1	---	0
Way D	0	---	0

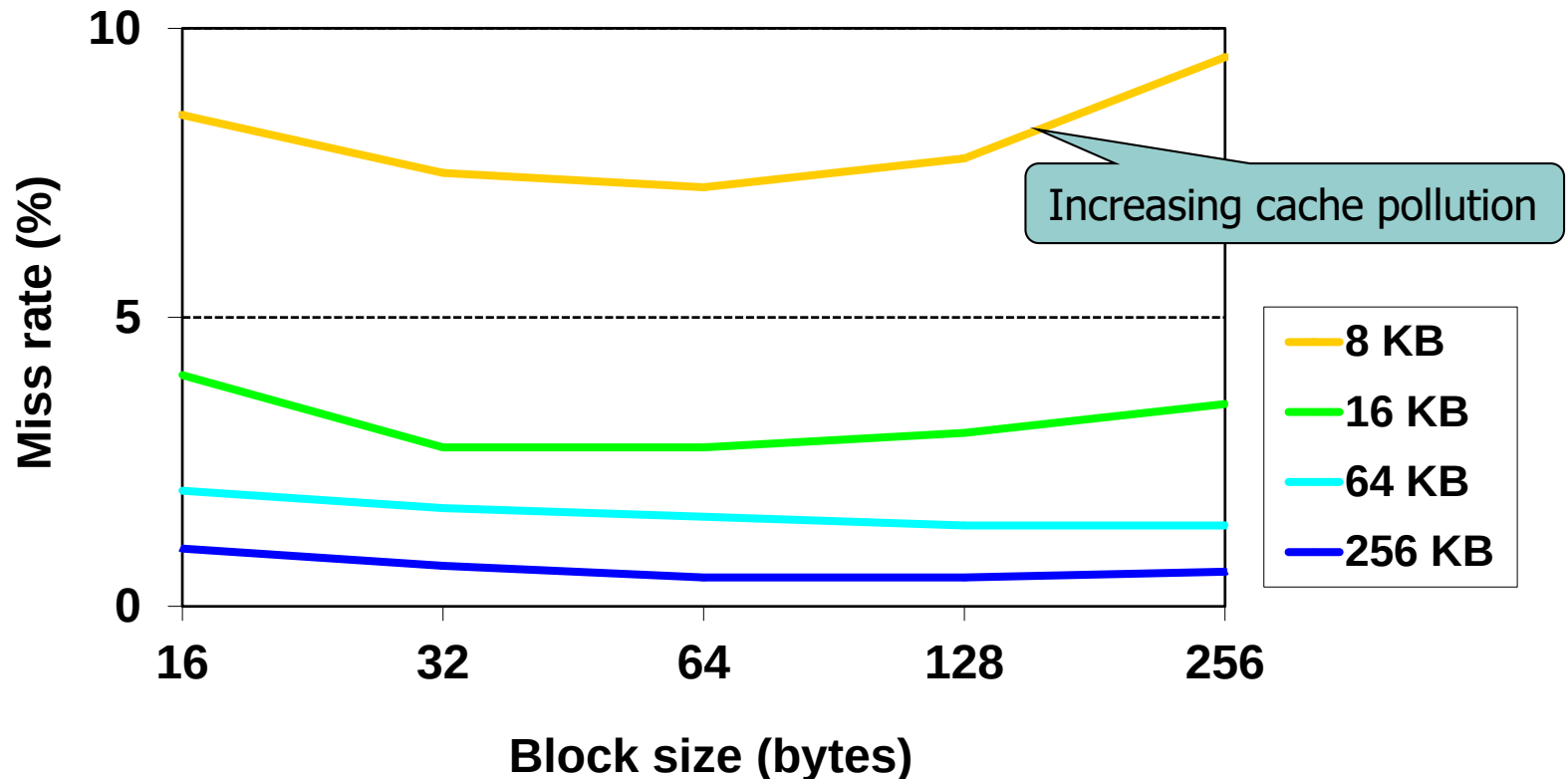
Replacement Decision

CD	AB	AB/CD	Way to replace
L2	L1	L0	
X	0	0	Way A
X	1	0	Way B
0	X	1	Way C
1	X	1	Way D

Not Recently Used (NRU)

- Use R(eferenced) and M(odified) bits
 - 0 (not referenced or not modified)
 - 1 (referenced or modified)
- Classify lines into
 - C0: R=0, M=0
 - C1: R=0, M=1
 - C2: R=1, M=0
 - C3: R=1, M=1
- Chose the victim from the lowest class
 - $(C3 > C2 > C1 > C0)$
- Periodically clear R and M bits

Miss Rate vs Block Size vs Cache Size

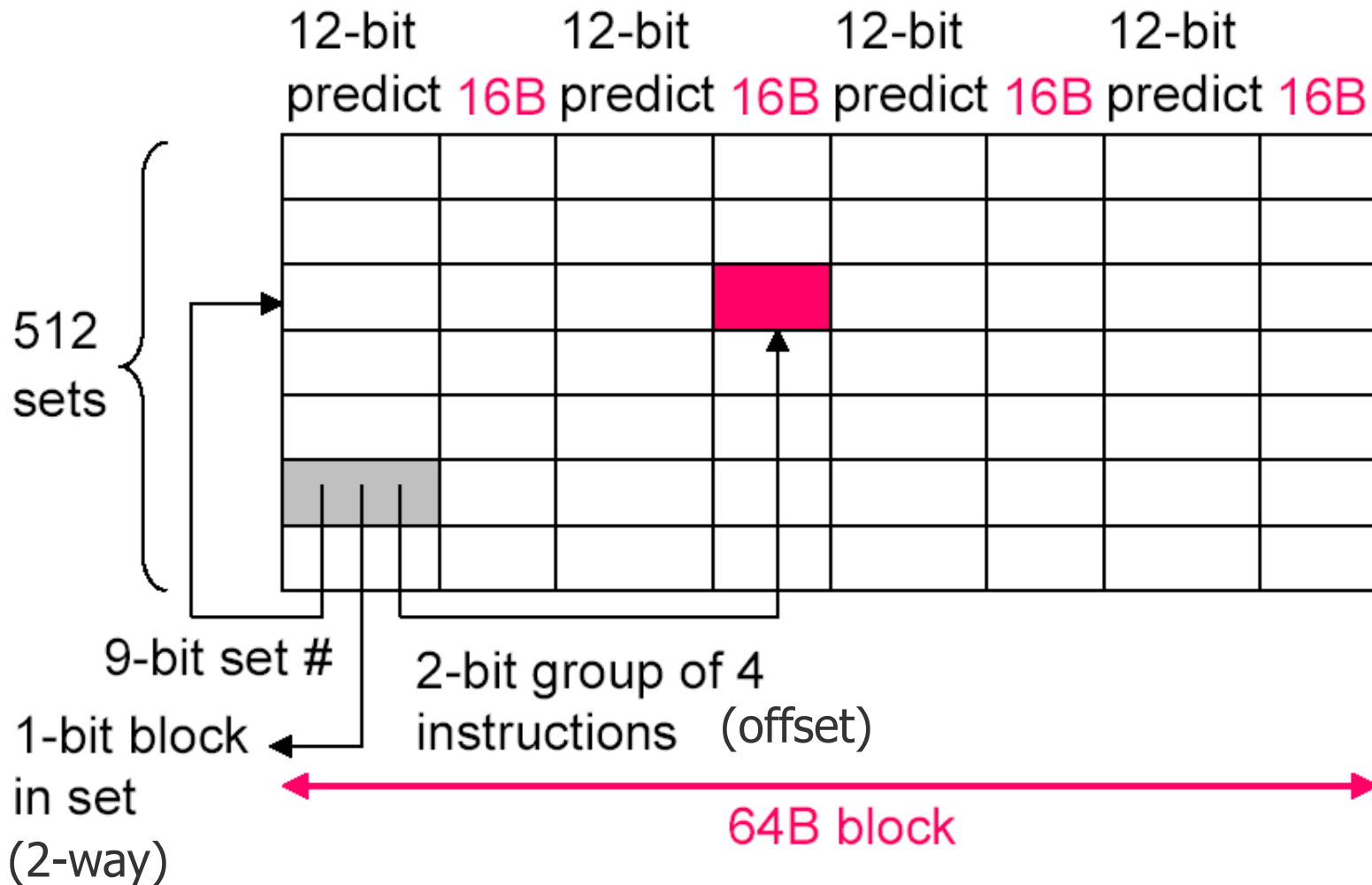


- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller
 - Stated alternatively, spatial locality among the words in a word decreases with a very large block; Consequently, the benefits in the miss rate become smaller

Reduce Miss Rate/Penalty: Way Prediction

- Best of both worlds: Speed as that of a DM cache and reduced conflict misses as that of a SA cache
- Extra bits predict the **way** of the next access
- Alpha 21264 Way Prediction (**next line predictor**)
 - If correct, 1-cycle I-cache latency
 - If incorrect, 2-cycle latency from I-cache fetch/branch predictor
 - Branch predictor can override the decision of the way predictor

Alpha 21264 Way Prediction



Note: Alpha advocates to align the branch targets on octaword (16 bytes)

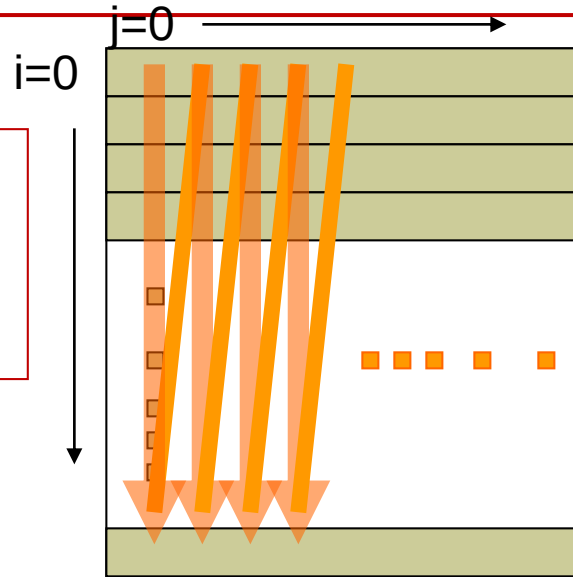
Reduce Miss Rate: Code Optimization

- Misses occur if sequentially accessed array elements come from different cache lines
- Code optimizations → No hardware change
 - Rely on programmers or compilers
- Examples:
 - Loop interchange
 - In nested loops: outer loop becomes inner loop and vice versa
 - Loop blocking
 - partition large array into smaller blocks, thus fitting the accessed array elements into cache size
 - enhances cache reuse

Loop Interchange

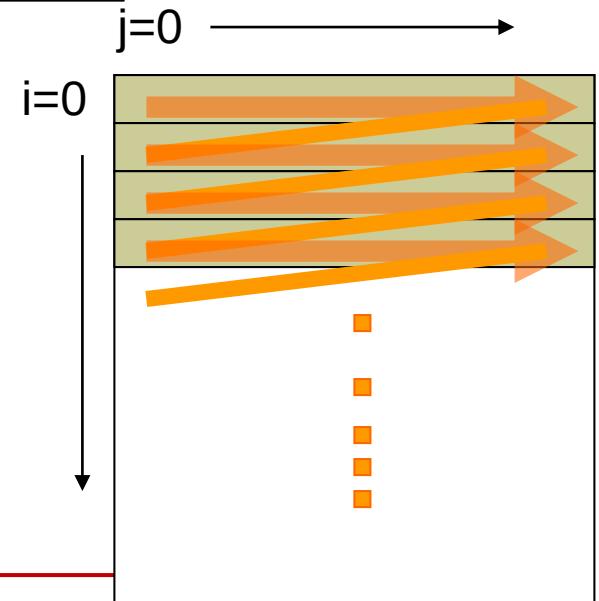
Row-major ordering

```
/* Before */  
for (j=0; j<100; j++)  
  for (i=0; i<5000; i++)  
    x[i][j] = 2*x[i][j]
```



What is the worst that could happen?

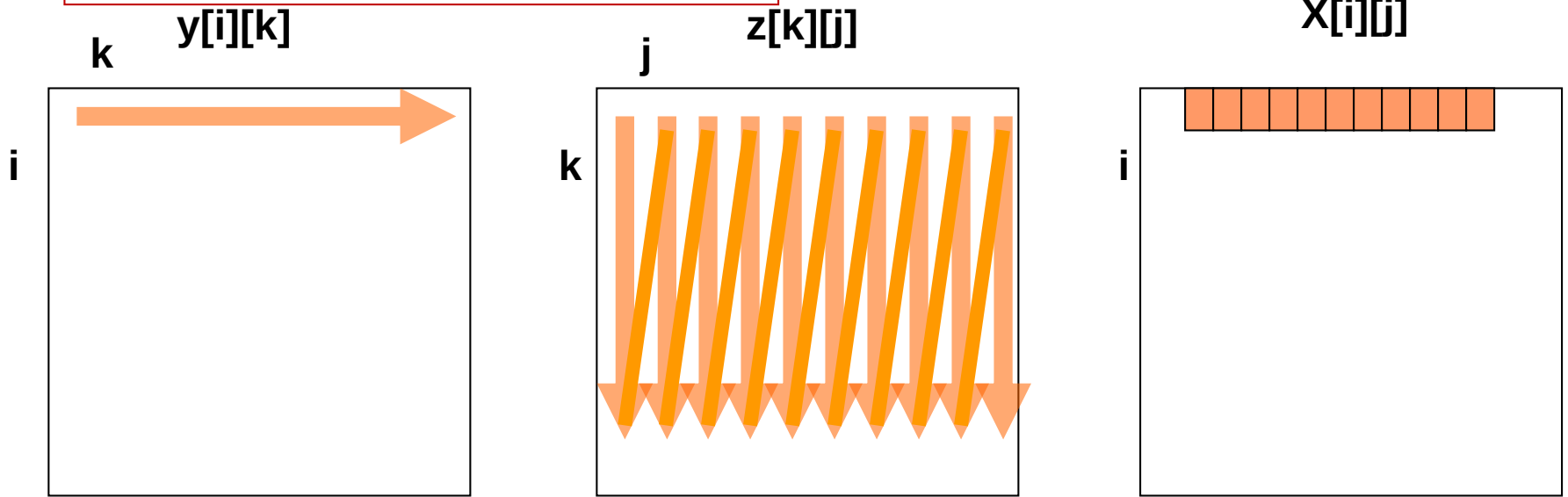
```
/* After */  
for (i=0; i<5000; i++)  
  for (j=0; j<100; j++)  
    x[i][j] = 2*x[i][j]
```



Improved cache efficiency

Loop Blocking

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

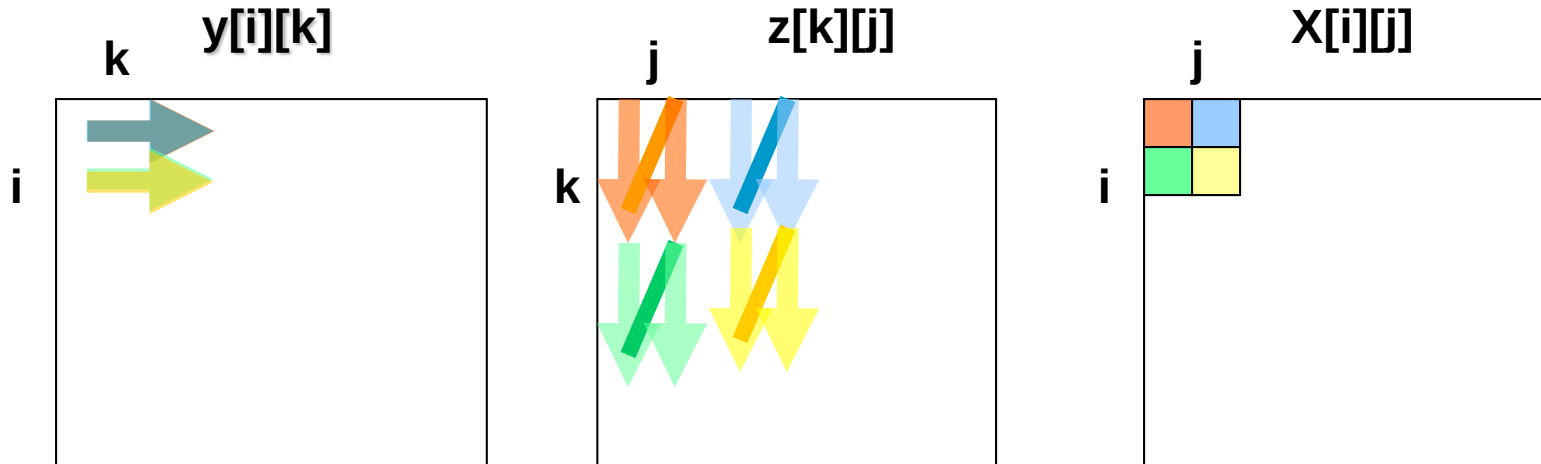


Does not exploit locality!

Loop Blocking

- Partition the loop's iteration space into many smaller chunks and ensure that the data stays in the cache until it is reused

```
/* After */  
for (jj=0; jj<N; jj=jj+B) // B: blocking factor  
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 += y[i][k]*z[k][j];  
        x[i][j] = x[i][j] + r;  
    }
```



Other Miss Penalty Reduction Techniques

- Critical value first and Restart early
 - Send requested data in the leading edge transfer
 - Trailing edge transfer continues in the background
- Give priority to read misses over writes
 - Use write buffer (WT) and writeback buffer (WB)
- Combining writes
 - combining write buffer
 - Intel's WC (write-combining) memory type
- Victim caches
- Assist caches
- Non-blocking caches
- Data Prefetch mechanism

Write Combining Buffer

- Need to initiate 4 separate writes back to lower level memory

Wr. addr	V	V	V	V
100	1	Mem[100]	0	0
108	1	Mem[108]	0	0
116	1	Mem[116]	0	0
124	1	Mem[124]	0	0

For WC buffer, combine neighbor addresses

- One single write back to lower level memory

Wr. addr	V	V	V	V				
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

WC memory type

- Intel 32 (starting in P6) supports USWC (or WC) memory type
 - Uncacheable, speculative Write Combining
 - Expensive (in terms of time) for individual write
 - Combine several individual writes into a bursty write
 - Effective for video memory data
 - Algorithm writing 1 byte at a time
 - Combine 32 of 1-byte data into one 32-byte write
 - Ordering is not important