EECS 645
Computer Architecture

Lecture 9 Memory Hierarchy Design – Part 1

Chenyun Pan

Department of Electrical Engineering &

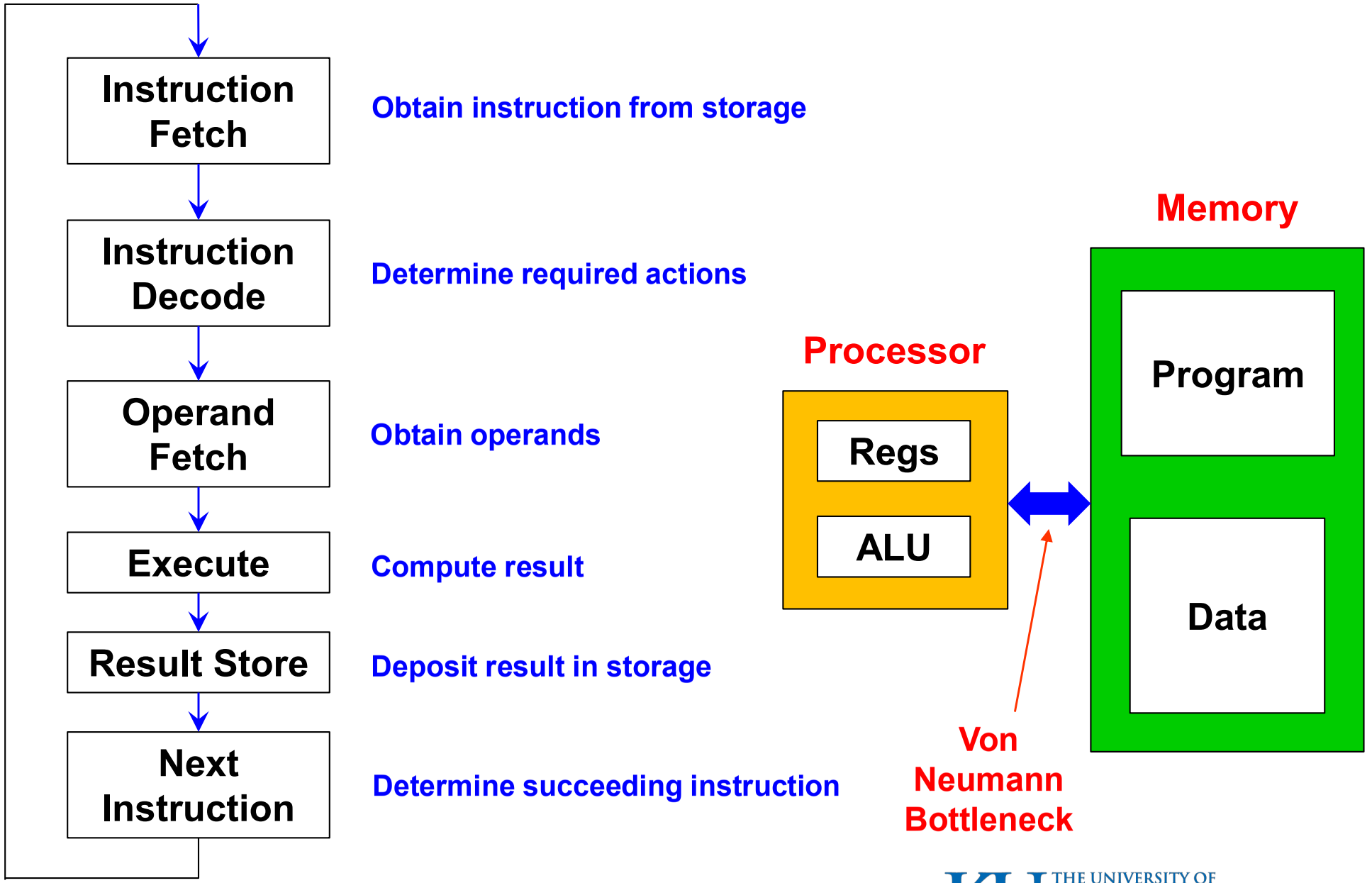
Computer Science

University of Kansas

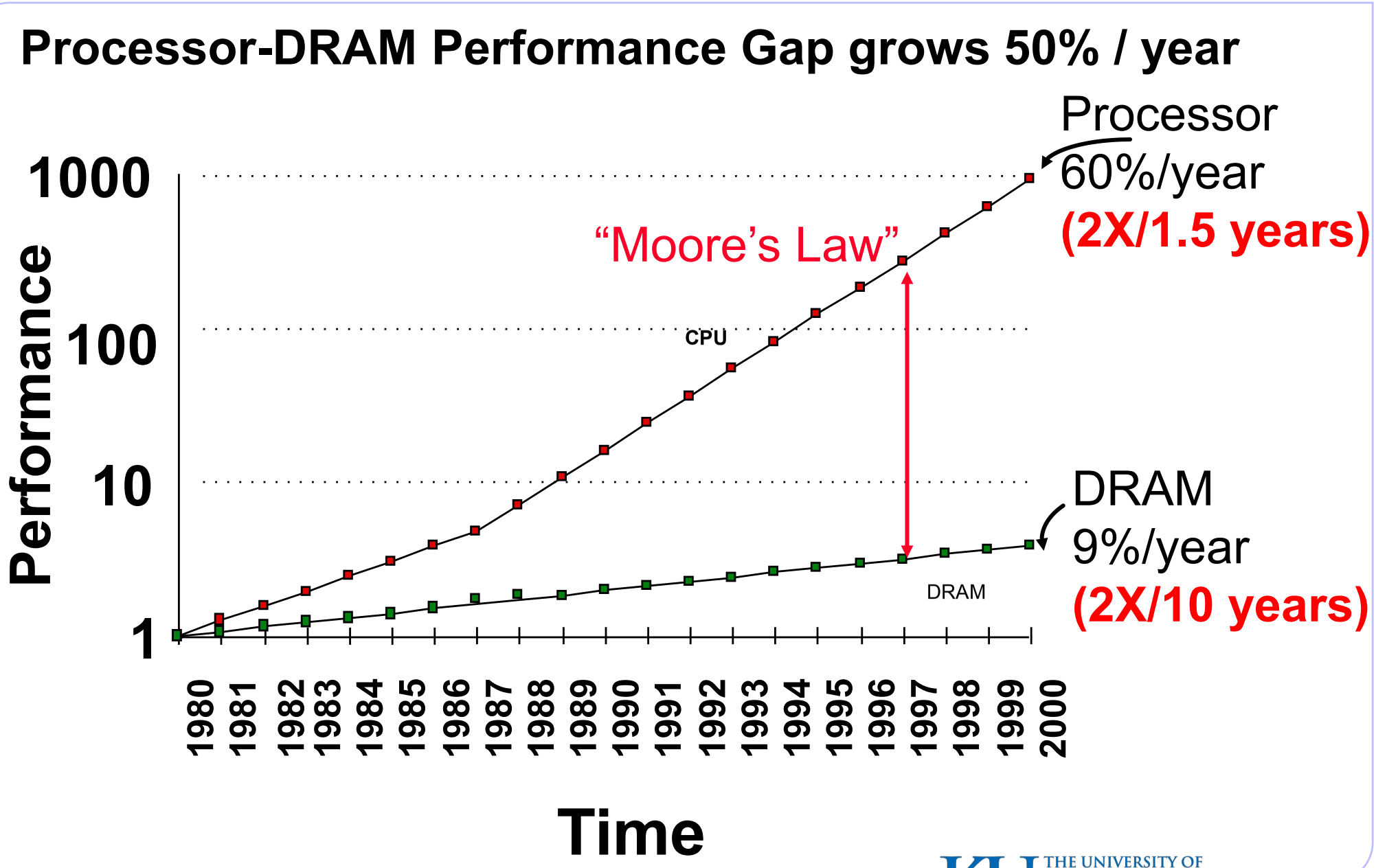
Slide Courtesy of Dr. Hsien-Hsin Sean Lee

KU THE UNIVERSITY OF KANSAS

Execution Style of von Neumann Machines



Why Care About Memory Hierarchy?



Energy Comparison

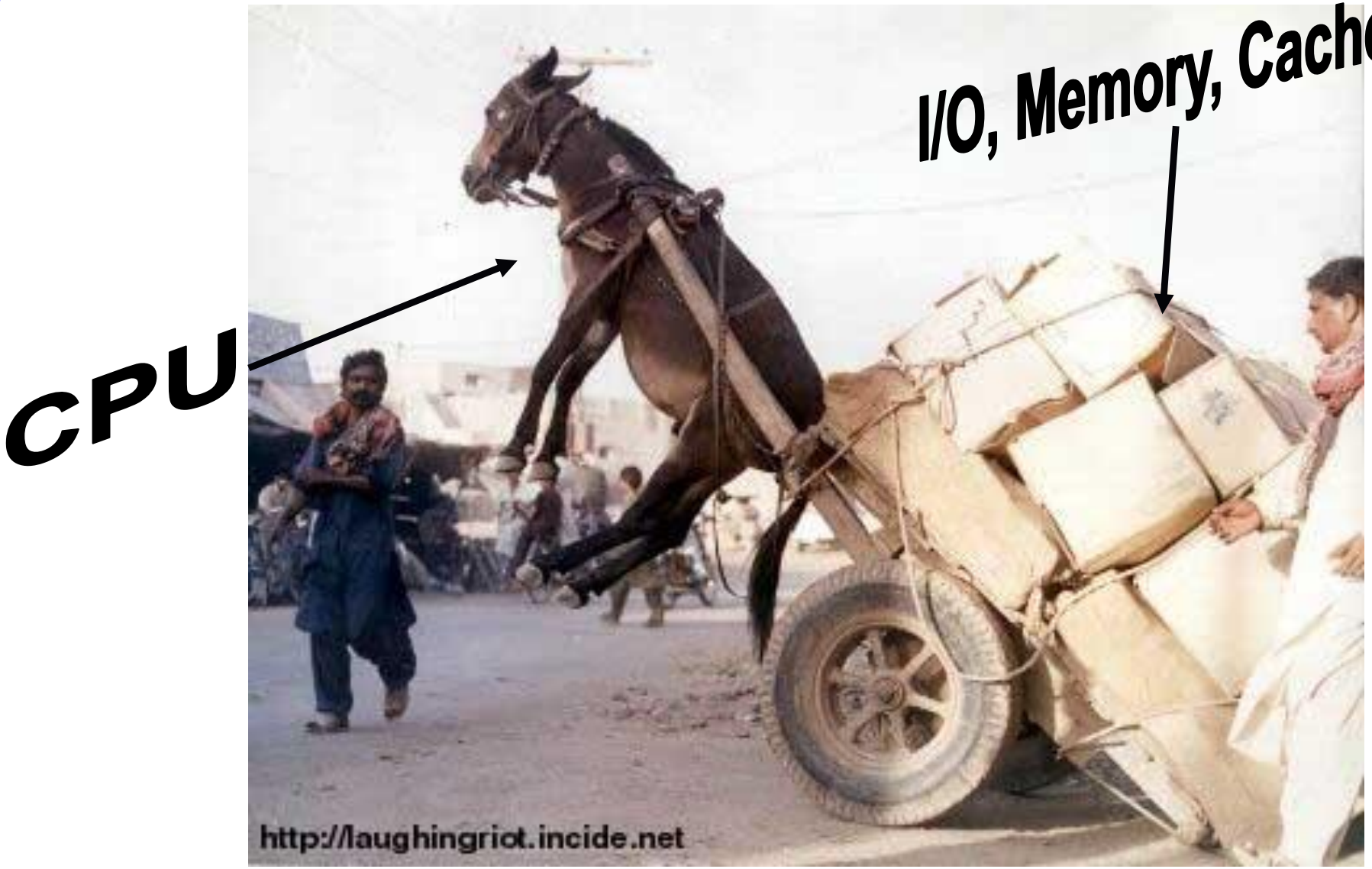
Energy table for 45 nm CMOS process [1]

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit DRAM	**640**	**6400**

DRAM access uses 100x – 1000x more energy than arithmetic operations

[1] M. Horowitz. Energy table for 45nm process, Stanford VLSI wiki.

An Unbalanced System

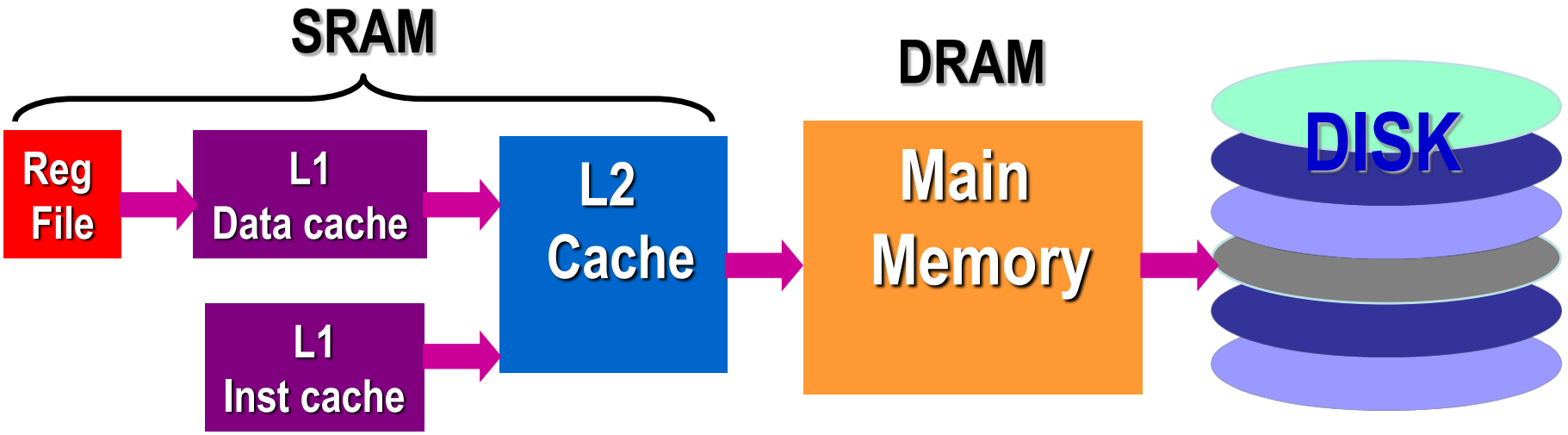


Source: Bob Colwell keynote ISCA'29 2002

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



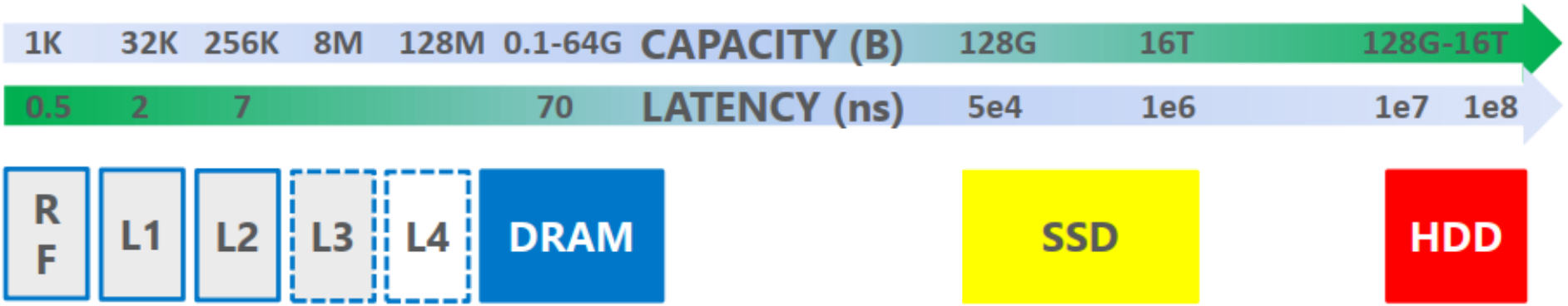
Memory Technology

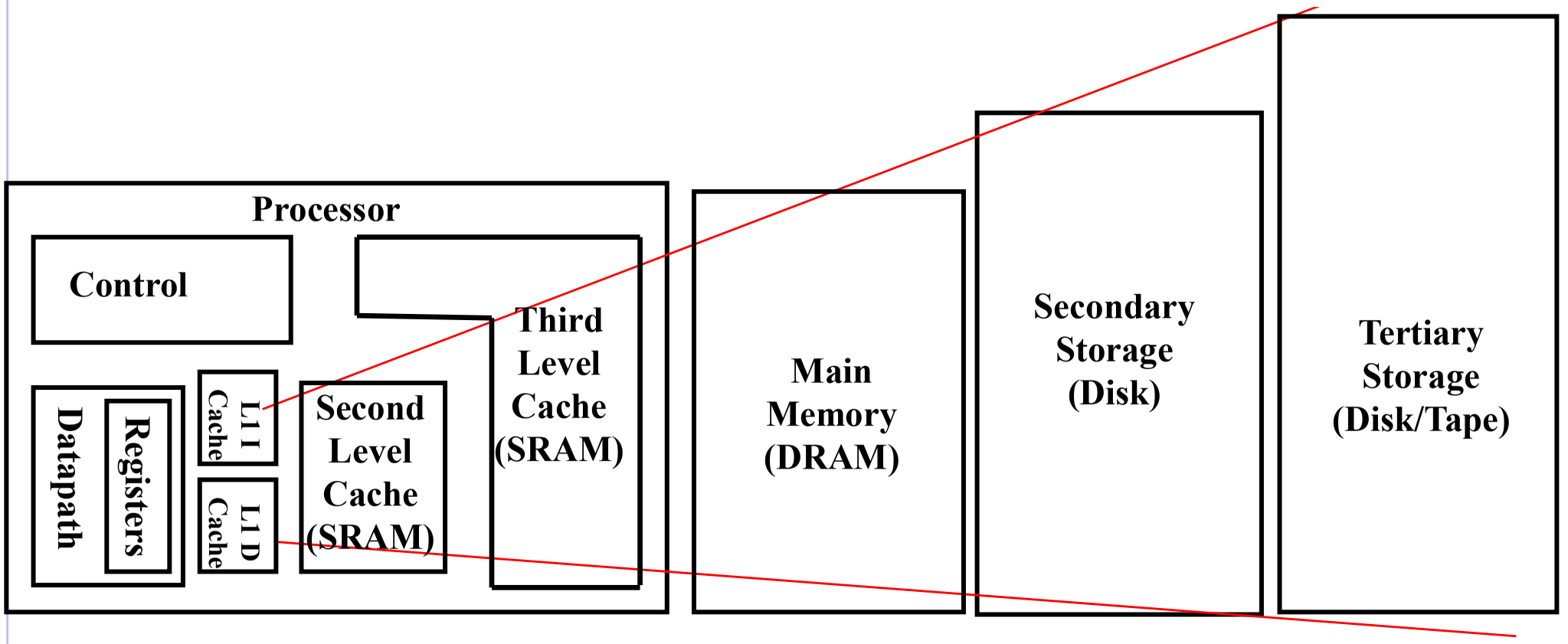
- Static RAM (SRAM)
  - ~1ns, ~$1000 per GB
- Dynamic RAM (DRAM)
  - ~50ns, ~$10 per GB
- Solid-State Drive (SSD)
  - ~100us, ~$0.2 per GB
- Magnetic disk
  - ~10ms, <$0.1 per GB

Ideal memory

- Access time of SRAM
- Capacity and cost/GB of disk

Modern Memory Hierarchy





Topics covered

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

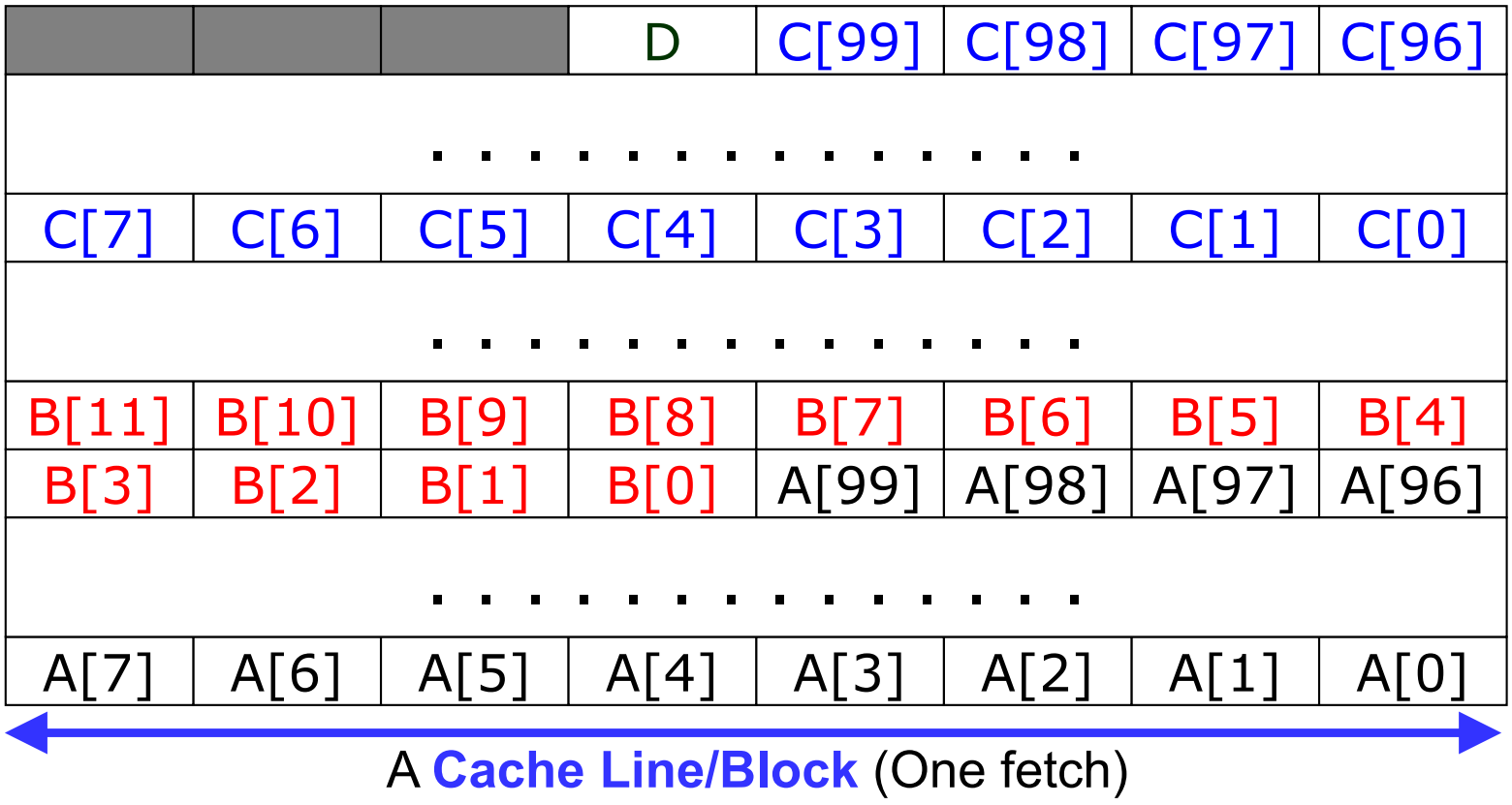
Principle of Locality

- Programs access a relatively small portion of address space at any instant of time.
- Two Types of Locality:
  - Temporal Locality (Locality in Time): If an address is referenced, it tends to be referenced again
    - e.g., loops, reuse
  - Spatial Locality (Locality in Space): If an address is referenced, neighboring addresses tend to be referenced
    - e.g., array access

Locality is a program property that is exploited in machine design.

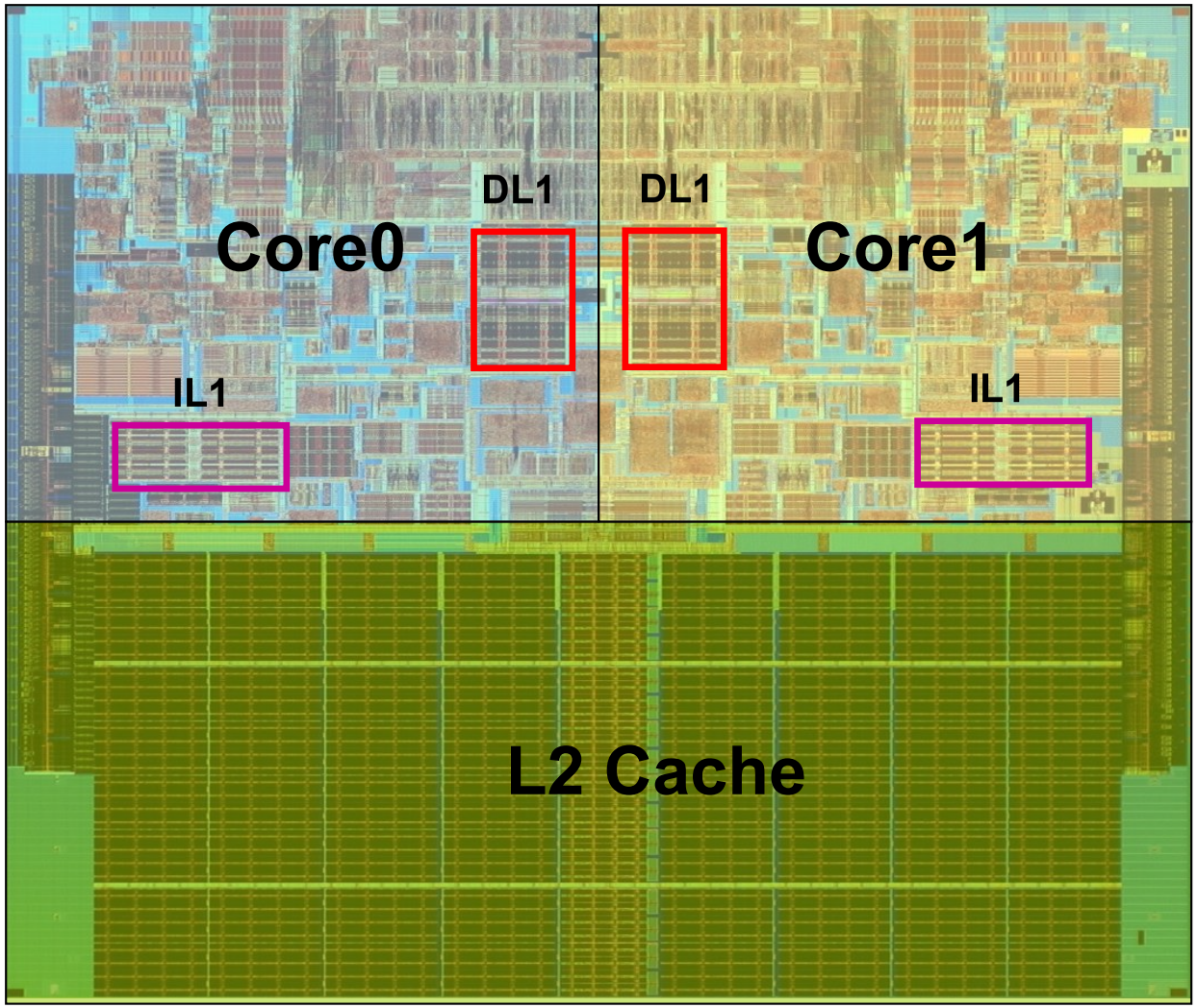
Example of Locality

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



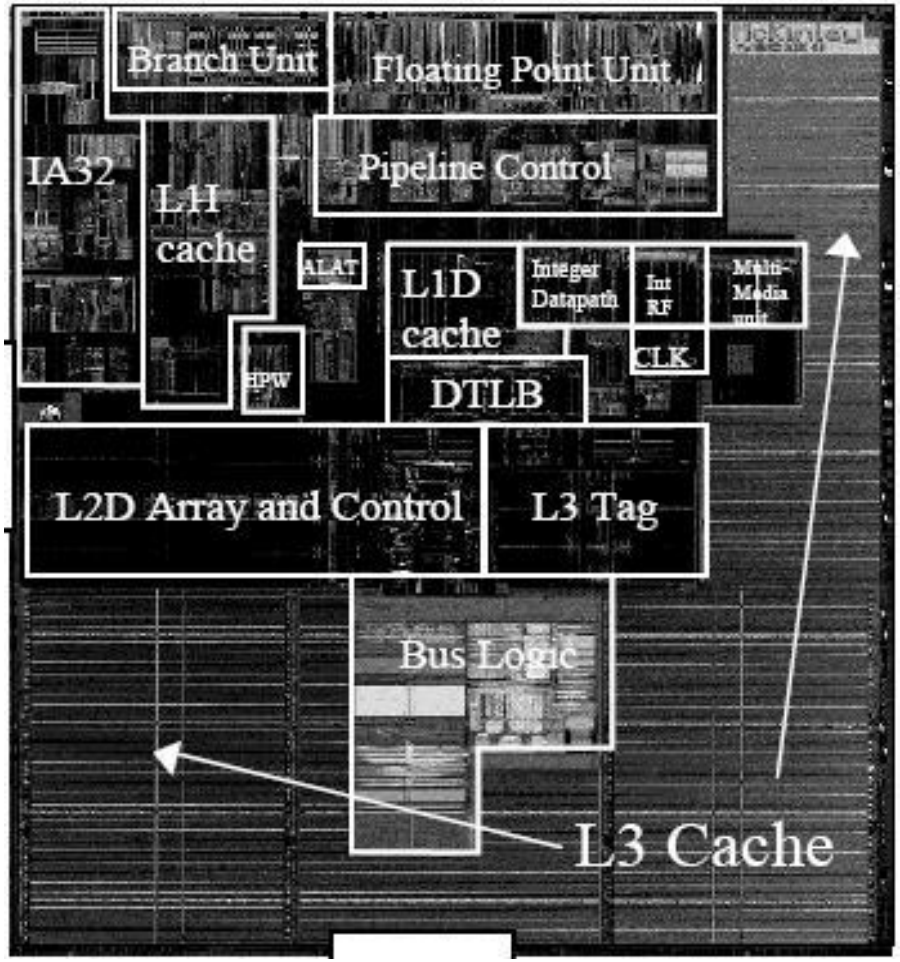
Example: Intel Core2 Duo

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

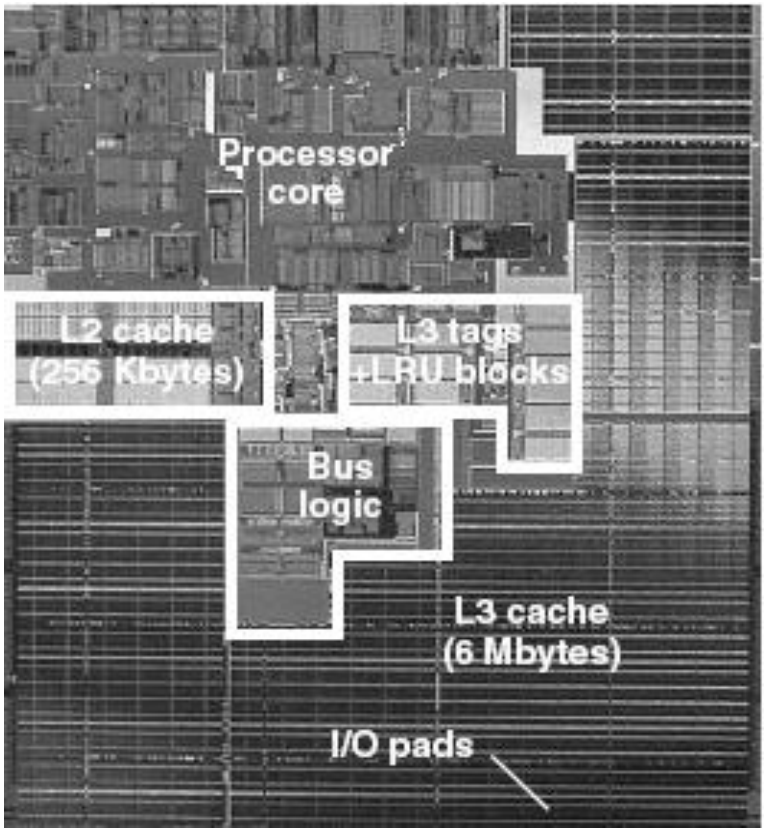


Source: http://www.sandpile.org

Example : Intel Itanium 2



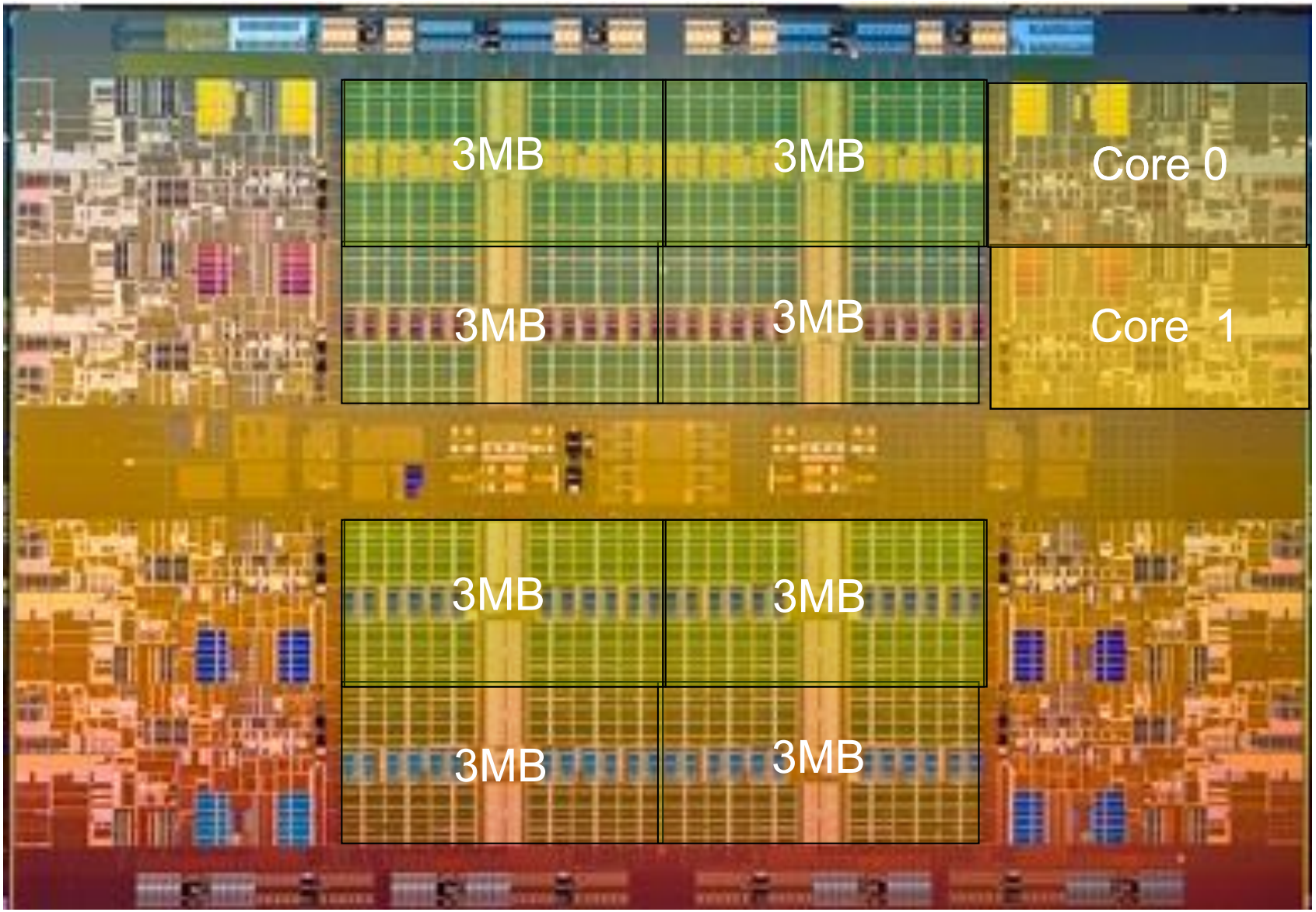
3MB
Version
180nm
421 mm²



6MB
Version
130nm
374 mm²

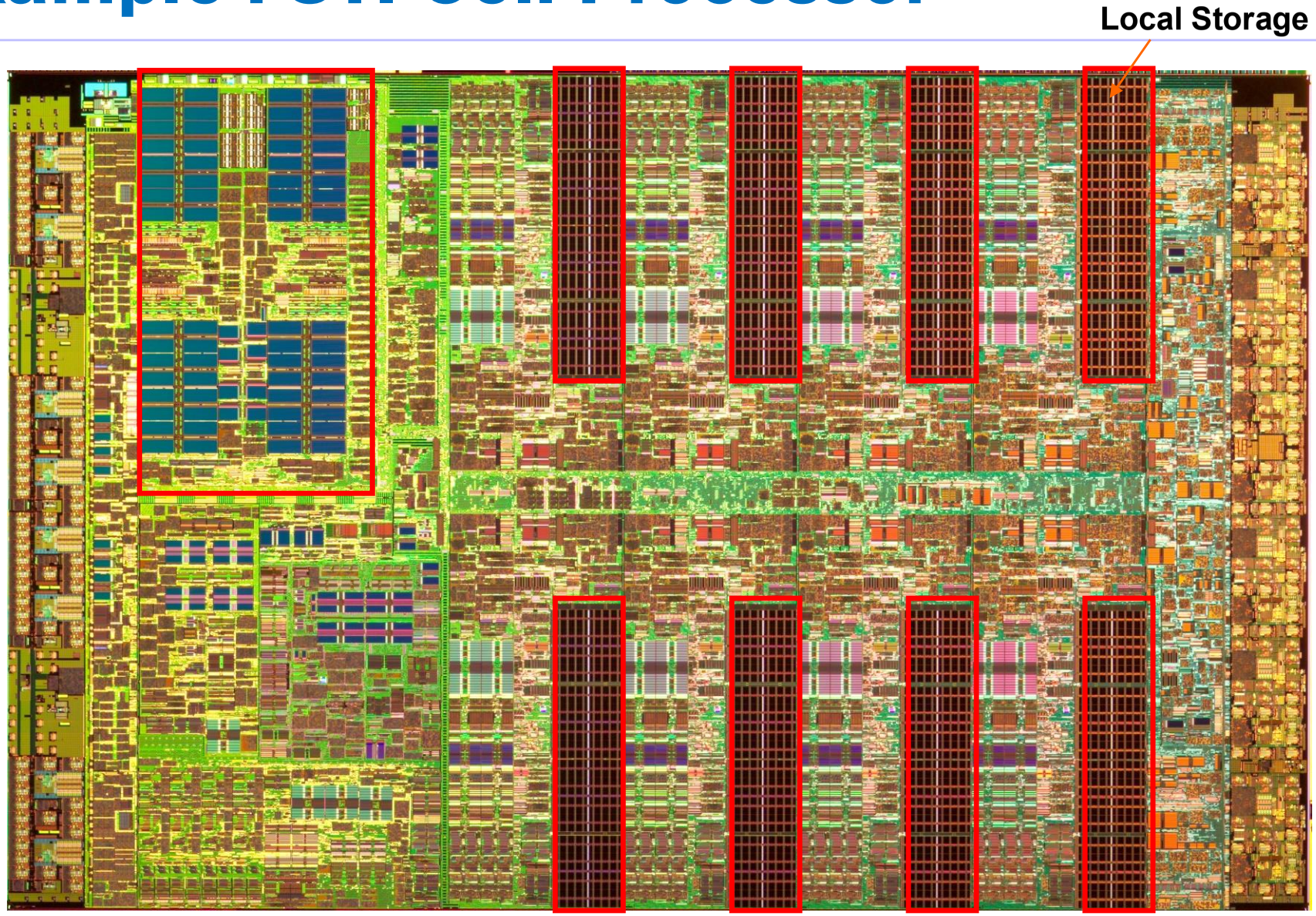
THE UNIVERSITY OF KANSAS

Intel Nehalem



24MB L3

Example : STI Cell Processor



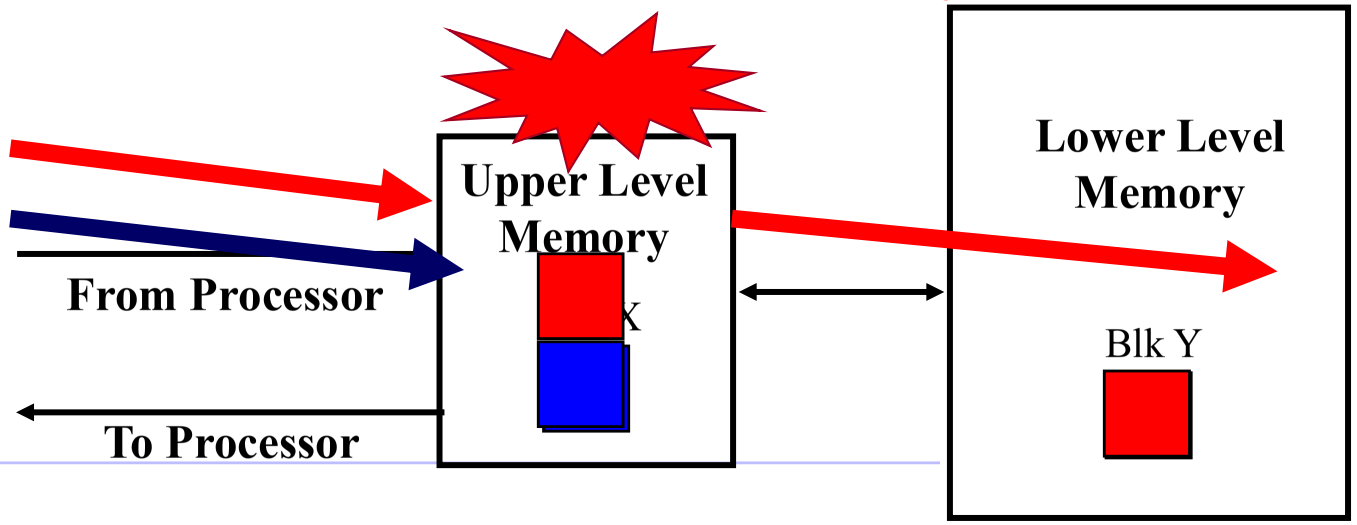
SPE = 21M transistors (14M array; 7M logic)

Topics covered

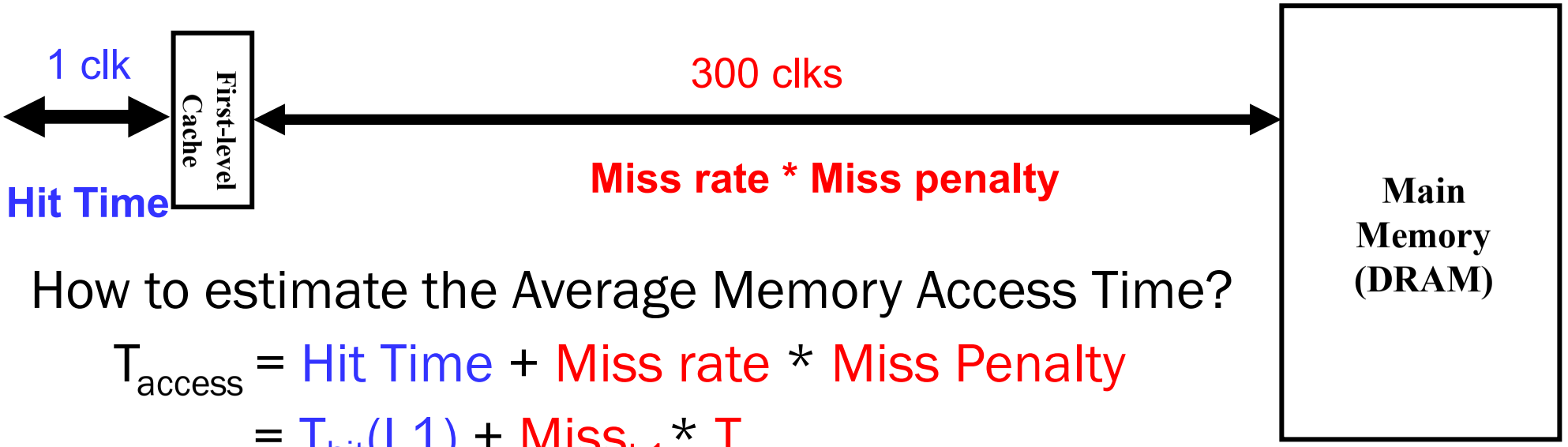
- Why do caches work
  - Principle of program locality
- Cache hierarchy
  - Average memory access time (T_{access})
- Types of caches
  - Direct mapped
  - Set-associative
  - Fully associative
- Cache policies
  - Write back vs. write through
  - Write allocate vs. No write allocate

Cache Terminology

- Cache Line/Block: amount of data being transferred at a time
- Hit: data appears in some line/block
  - Hit Rate: the fraction of memory accesses found in the level
  - Hit Time: Time to access the level (including the time to check if hit)
- Miss: data needs to be retrieved from a line/block in the lower level memory (e.g.DRAM)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a line/block in the upper level cache + Time to deliver the line/block to the processor
- Hit Time should be much shorter than Miss Penalty



Memory Hierarchy Performance



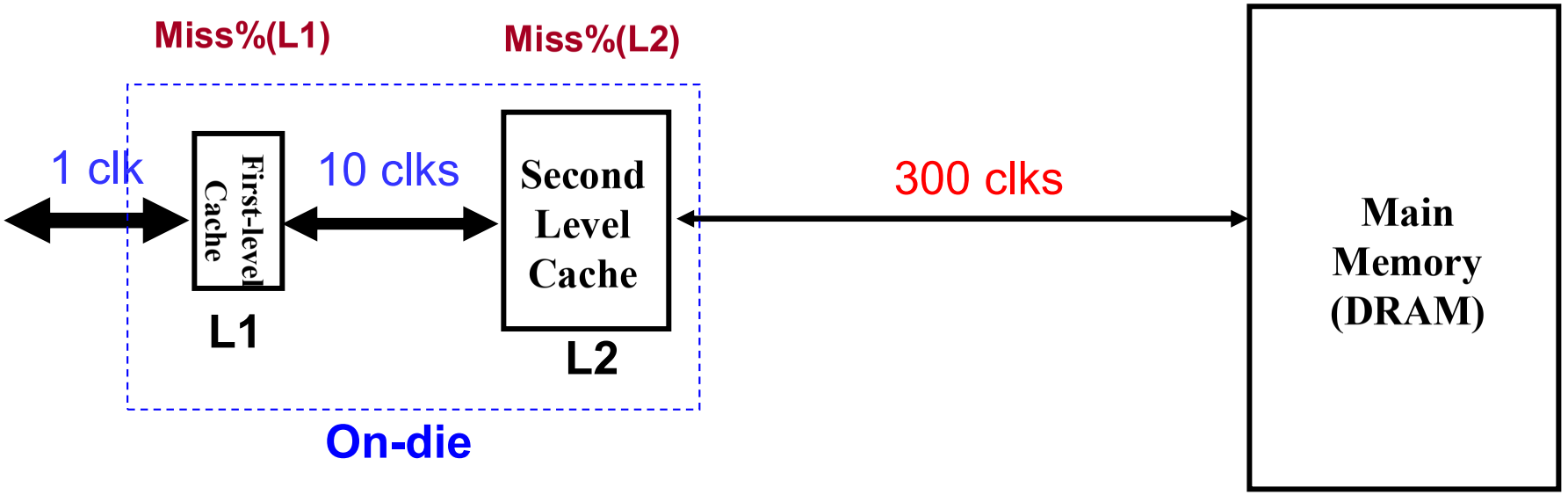
How to estimate the Average Memory Access Time?

T_{access} = Hit Time + Miss rate * Miss Penalty

= $T_{hit}(L1)$ + $Miss_{L1}$ * T_{mem}

- Example:
  - Cache Hit = 1 cycle
  - Miss rate = 10%
  - Miss penalty = 300 cycles
  - Access time = 1 + 0.1 * 300 = 31 cycles
  - 9.7X Speed Up!
- How to further improve it?

Reducing Penalty: Multi-Level Cache

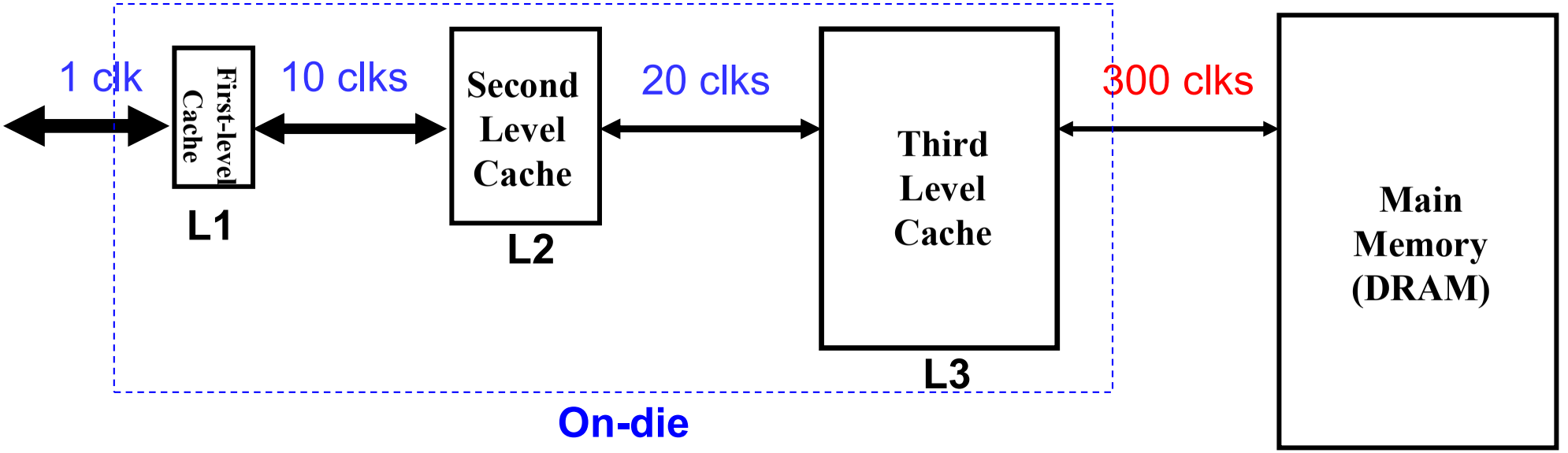


What is the Average Memory Access Time ?

$$T_{access} = T_{L1} + Miss_{L1} \cdot (T_{L2} + Miss_{L2} \cdot T_{mem})$$

A larger cache has a lower miss rate but a longer hit time.

Reducing Penalty: Multi-Level Cache



Average Memory Access Time

$$T_{access} = T_{L1} + Miss_{L1} \cdot [T_{L2} + Miss_{L2} \cdot (T_{L3} + Miss_{L3} \cdot T_{mem})]$$

T_{access} Example

- 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
- T_{access} = ?
  - 2.115 (compare to 31 with only L1 cache)

14.7x speed-up!

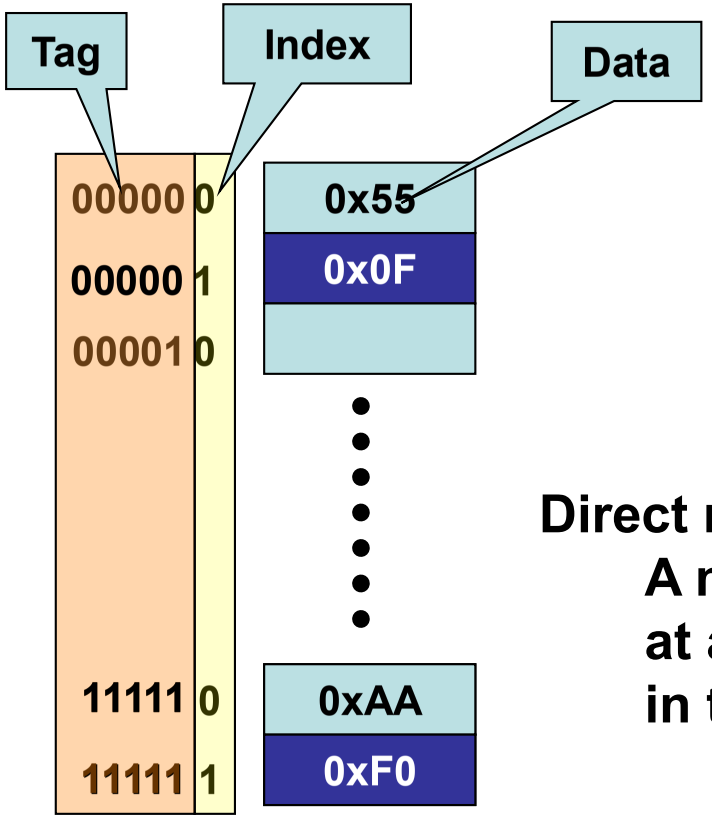
Topics covered

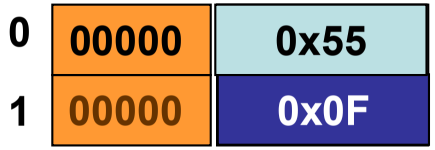
- Why do caches work
  - Principle of program locality
- Cache hierarchy
  - Average memory access time (T_{access})
- Types of caches
  - Direct mapped
  - Set-associative
  - Fully associative
- Cache policies
  - Write back vs. write through
  - Write allocate vs. No write allocate

Types of Caches

Type of cache	Mapping of data from memory to cache	Complexity of searching the cache
Direct mapped (DM)	A memory value can be placed at **a single corresponding location** in the cache	Fast indexing mechanism
Set-associative (SA)	A memory value can be placed in **different locations of a set** 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

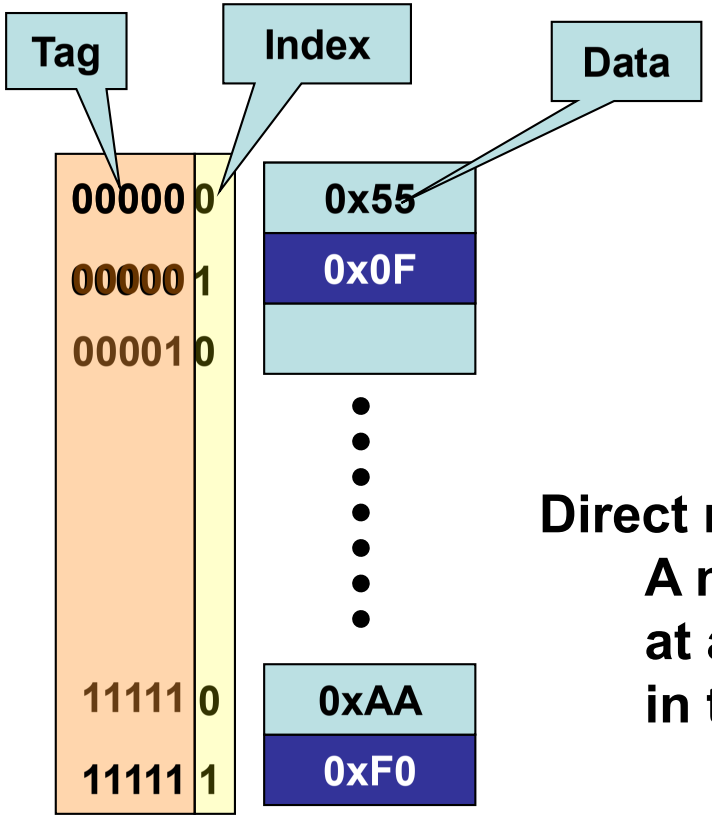


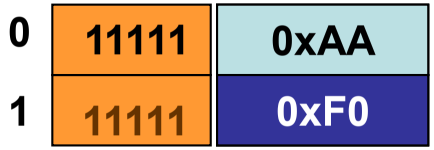


Direct mapping:

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

Direct Mapping

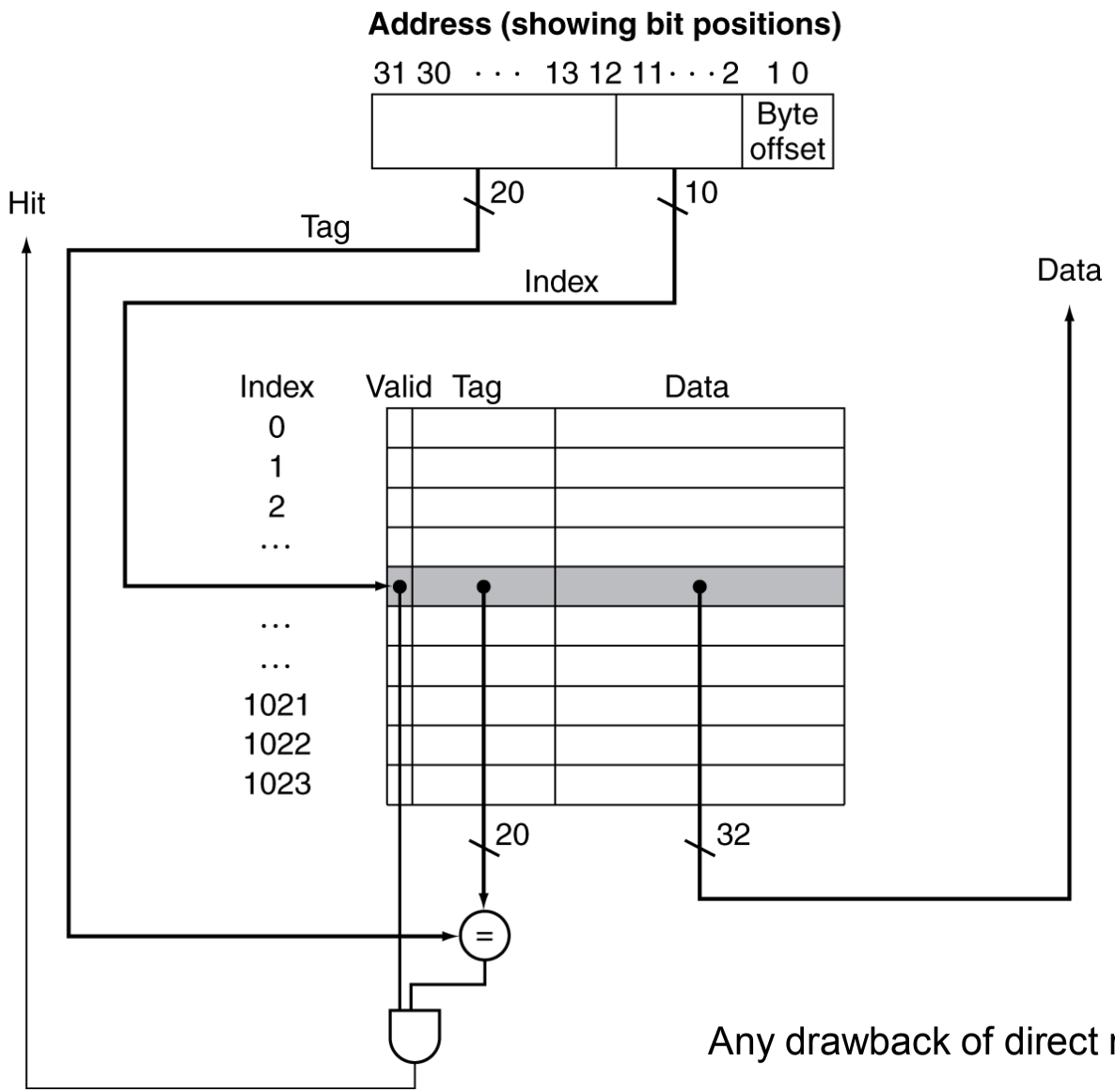




Direct mapping:

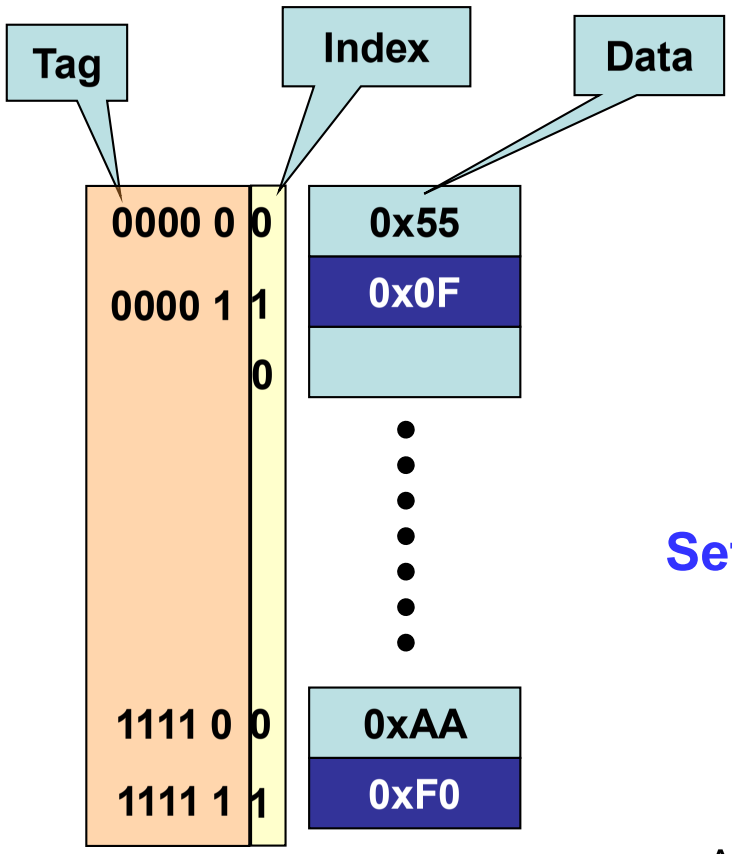
A memory value can only be placed at a single corresponding location in the cache

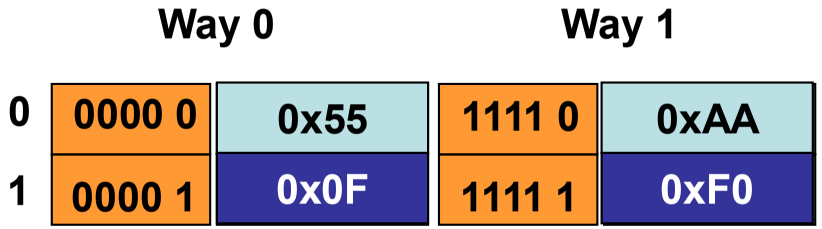
Address Subdivision



Any drawback of direct map cache?

Set Associative Mapping (2-Way)

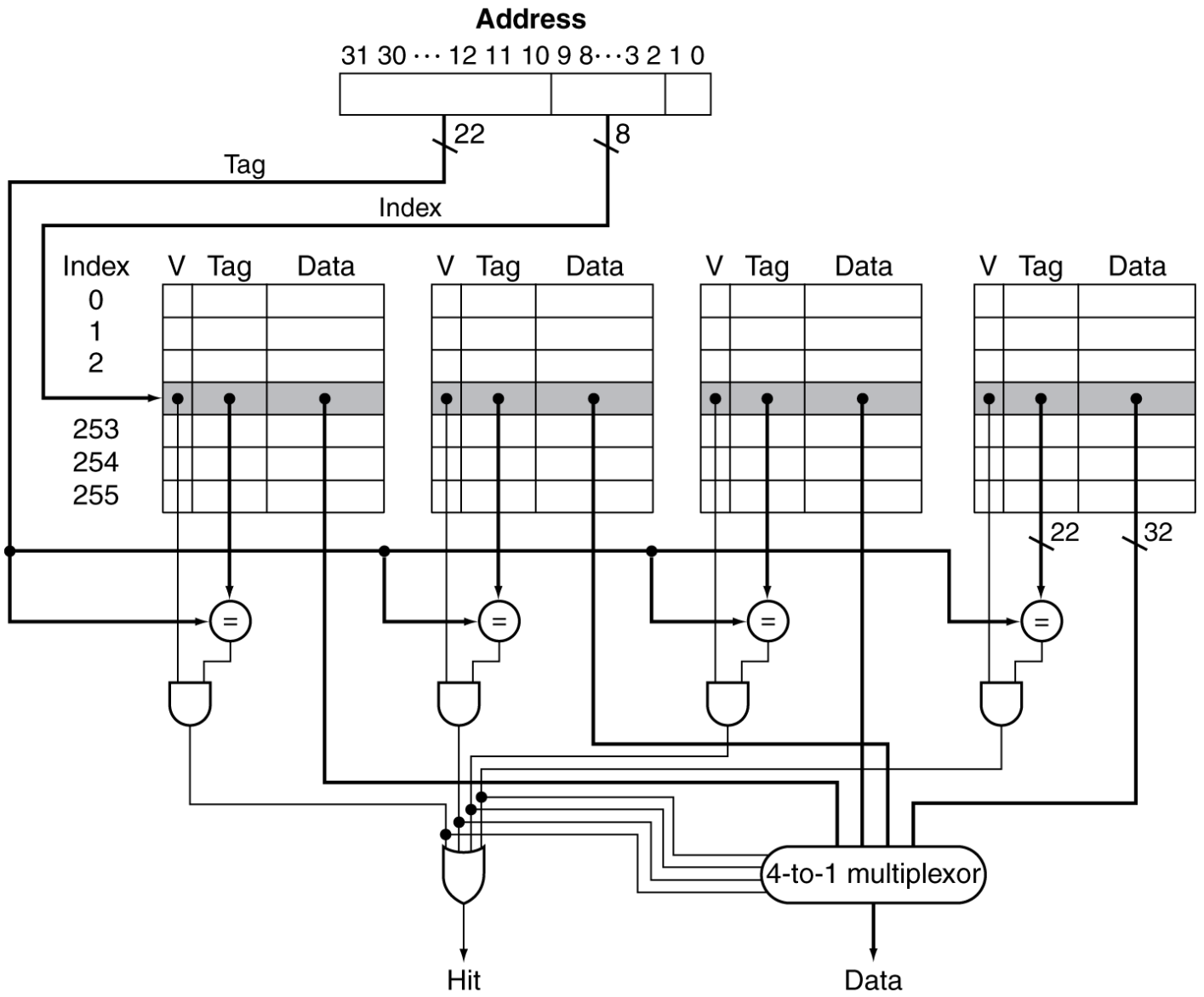




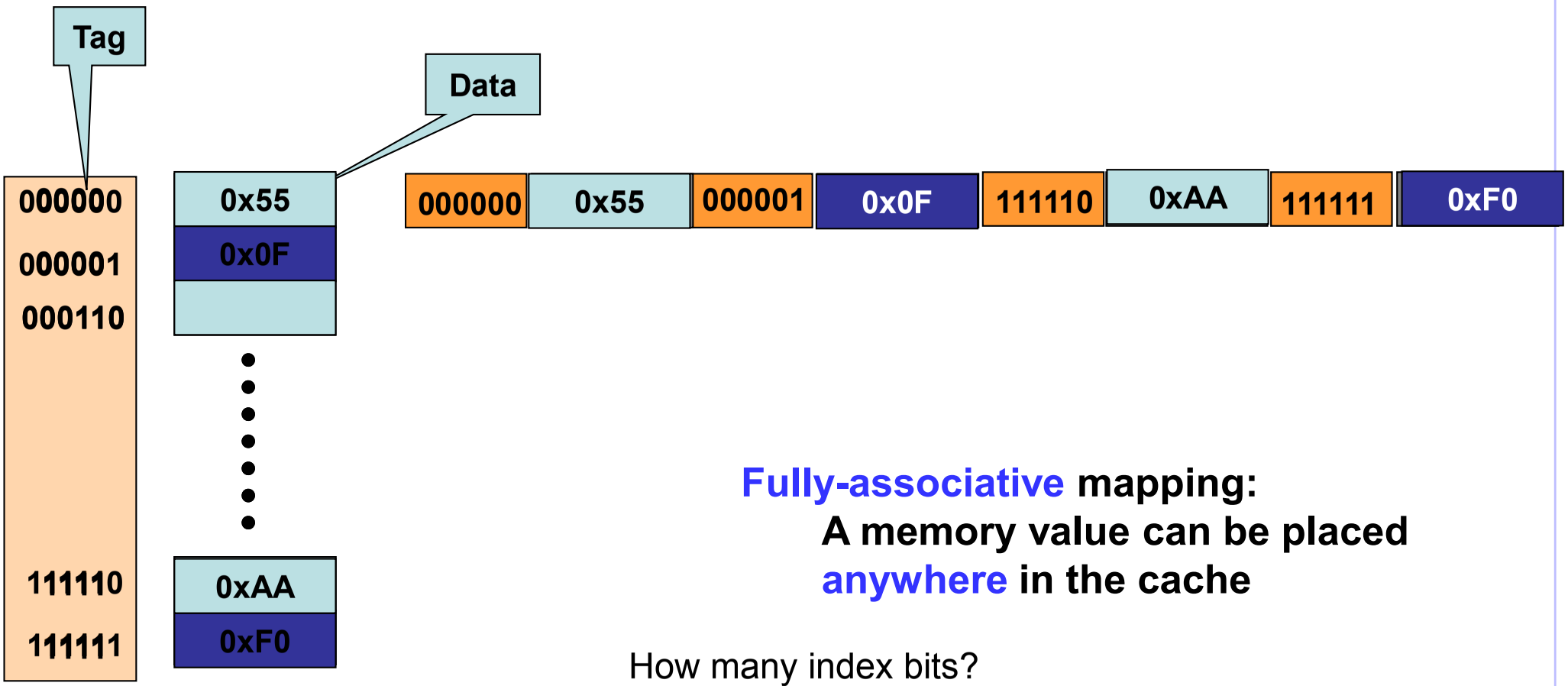
Set-associative mapping:
A memory value can be placed in multiple location of a set in the cache

Any drawback of set-associative map cache?

4-Way Set Associative Cache Organization



Fully Associative Mapping



Fully-associative mapping:
A memory value can be placed anywhere in the cache

How many index bits?
- None

What are the advantage and disadvantage of fully-associative map cache?

Cache Replacement Policy

- Random
  - Replace a randomly chosen line
- FIFO
  - Replace the oldest line
- LRU (Least Recently Used)
  - Replace the least recently used line

How Much Associativity

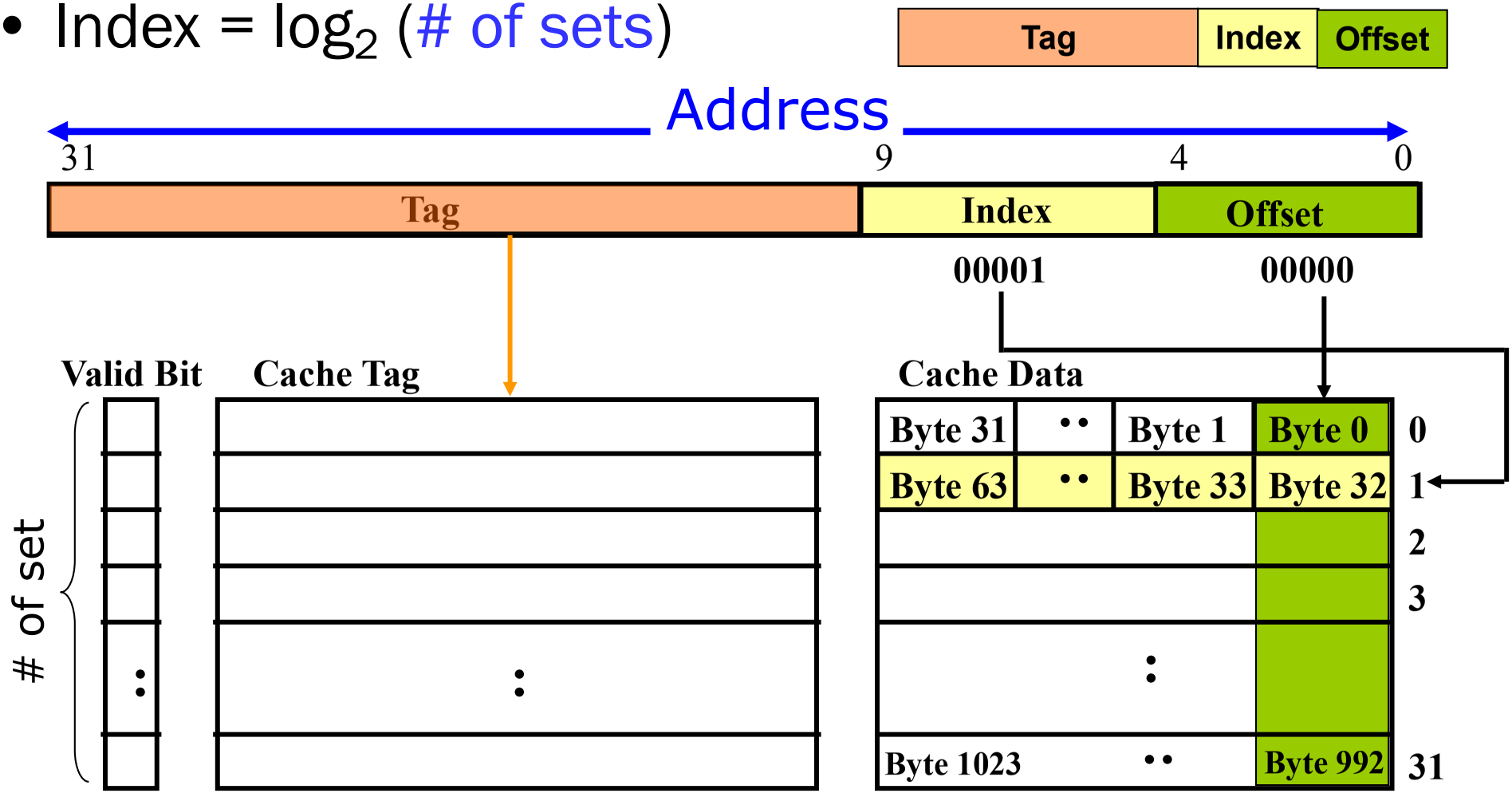
- Increased associativity decreases miss rate
- Simulation of a system with 64KB D-cache, 32-byte blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

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 (next lecture)

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 32-bit physical address
- Tag size?
  - # of lines = 2MB/64B = $2^{21}/2^{6} = 2^{15}$
  - Direct-mapped => # of sets = # of lines = 2^{15} => Index = 15 bits
  - Tag = 32 – 15 – 6 = 11 bits

- Now change it to 16-way, Tag size?
  - # of lines = 2MB/64B = $2^{21}/2^{6} = 2^{15}$
  - 16-way per set => # of sets = # of lines/16 = 2^{11} => Index = 11 bits
  - Tag = 32 – 11 – 6 = 15 bits

- How about if it's fully associative, Tag size?
  - # of sets = 1 => No Index
  - Tag = 32 – 6 = 26 bits

Topics covered

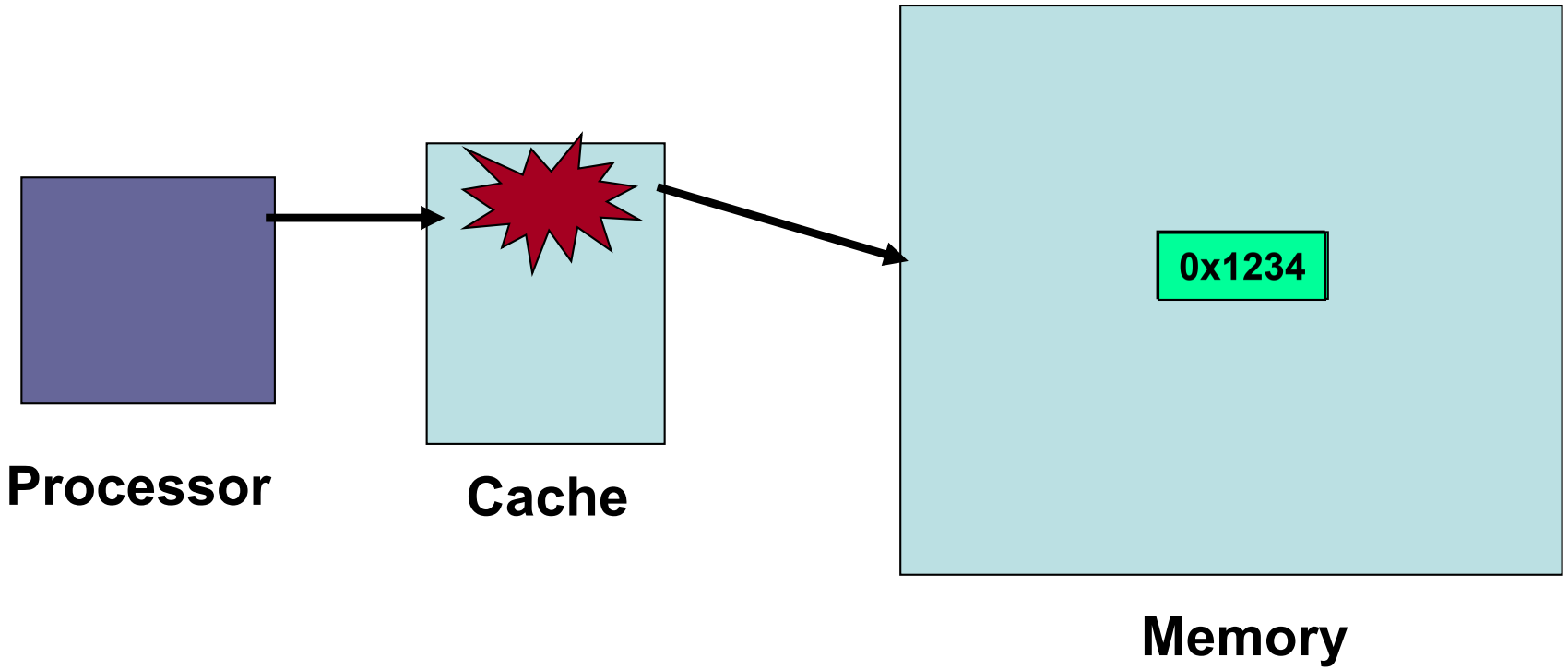
- Why do caches work
  - Principle of program locality
- Cache hierarchy
  - Average memory access time (T_{access})
- Types of caches
  - Direct mapped
  - Set-associative
  - Fully associative
- Cache policies
  - Write back vs. write through
  - Write allocate vs. No write allocate

Cache Write Policy

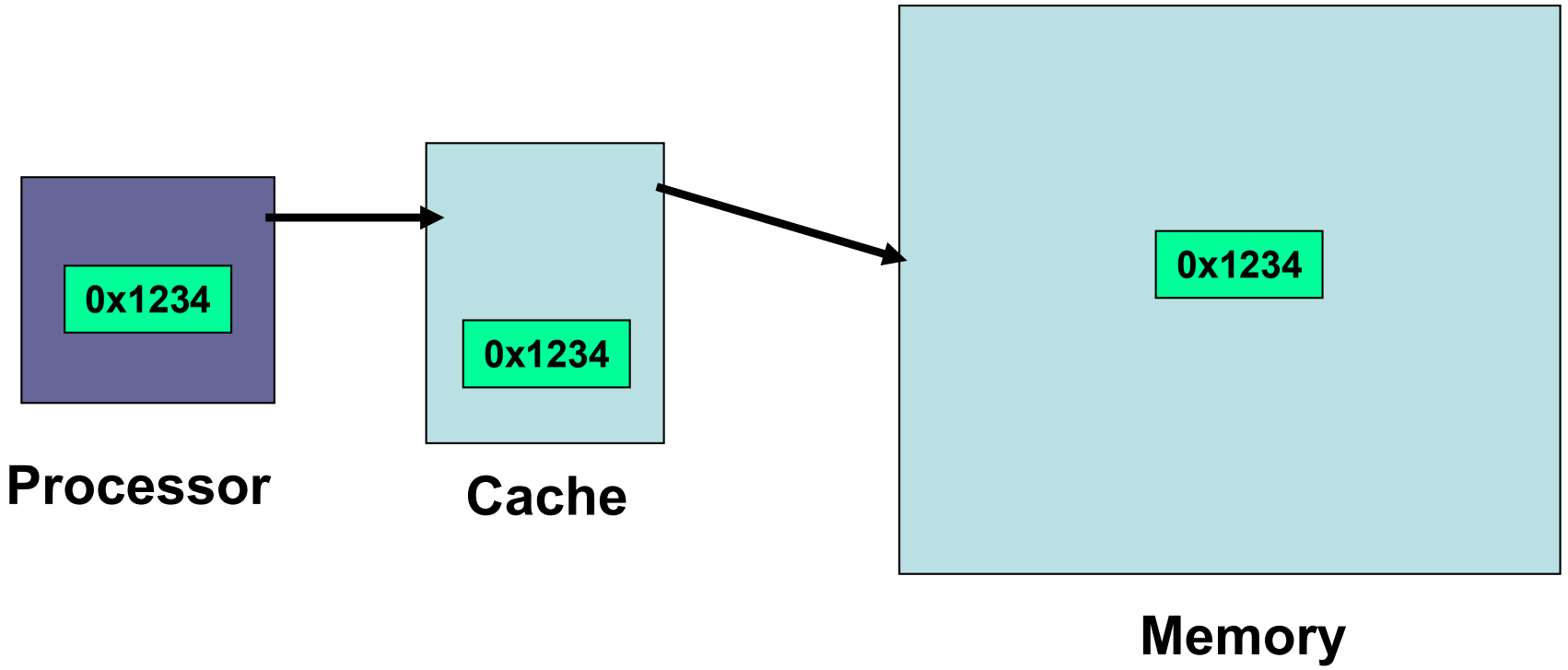
- *Write through* -The value is written to both the cache line and to the lower-level memory (e.g. main memory).

- *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.
  - Cache line is clean: holds the same value as memory
  - Cache line is dirty: holds a different value than memory

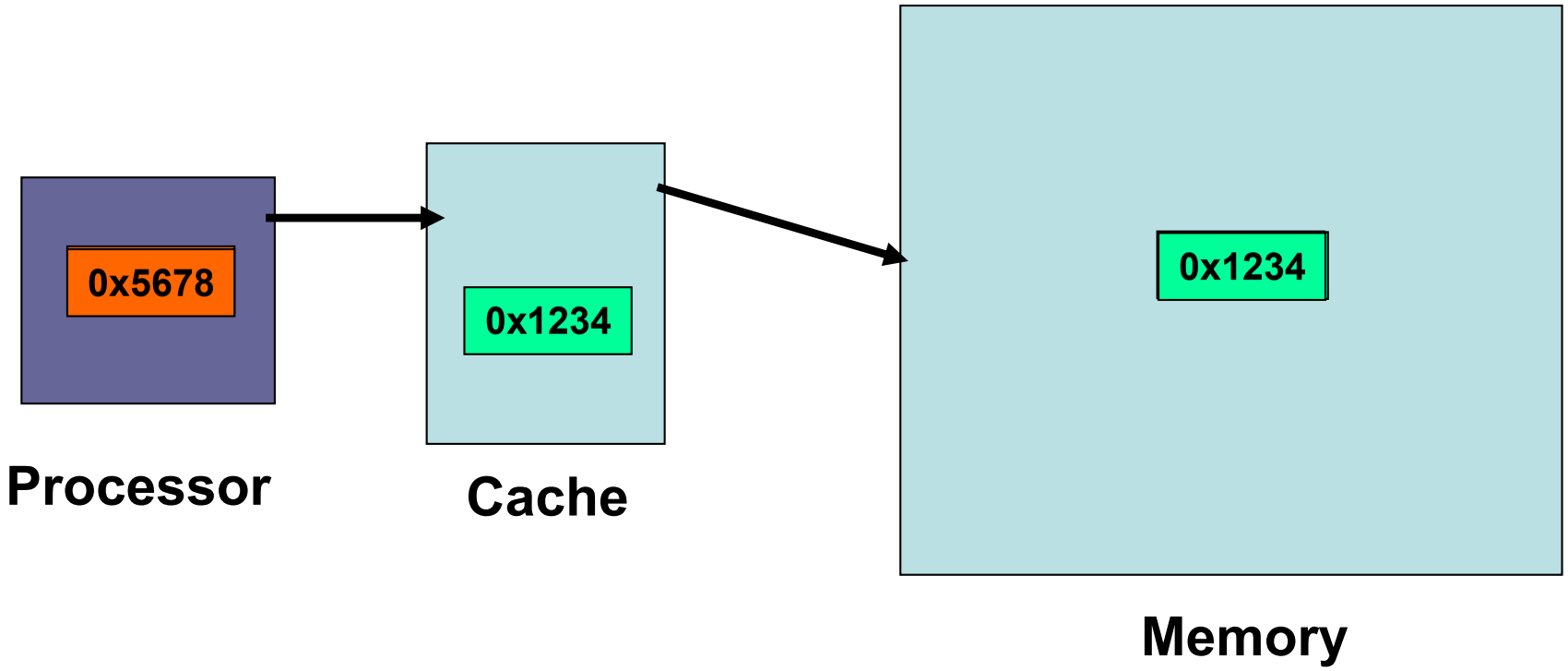
Write-through Policy



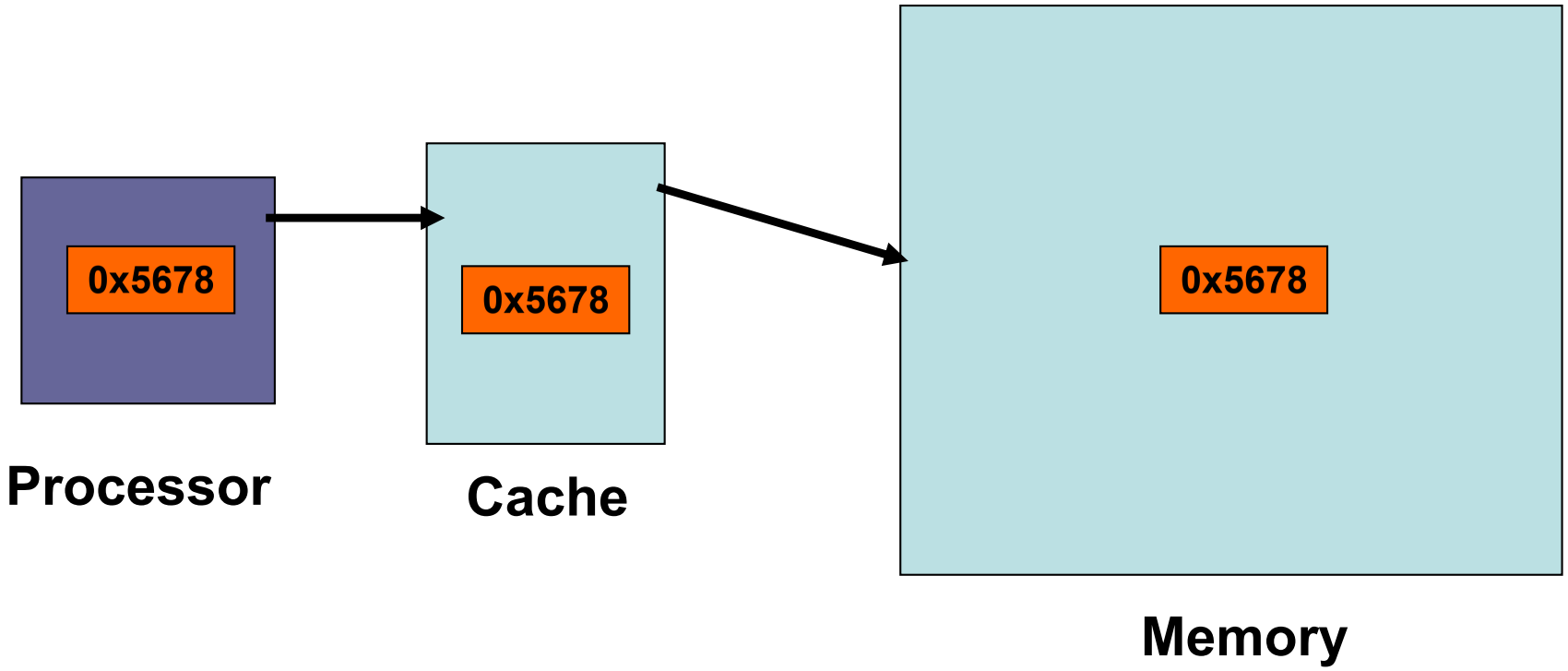
Write-through Policy



Write-through Policy

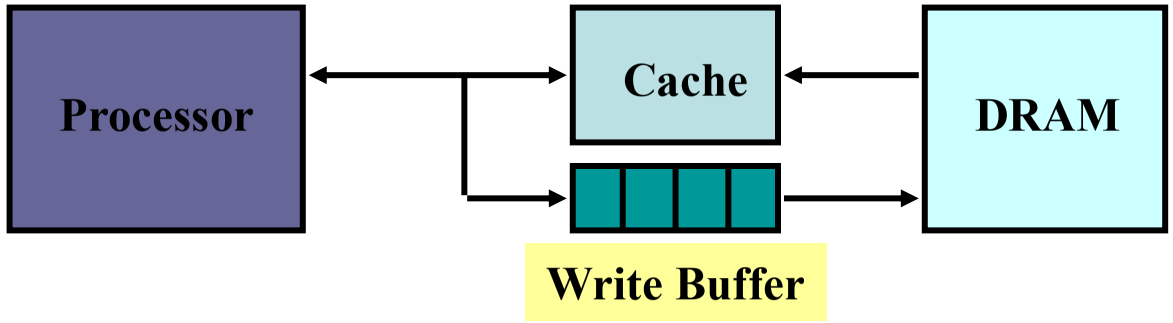


Write-through Policy



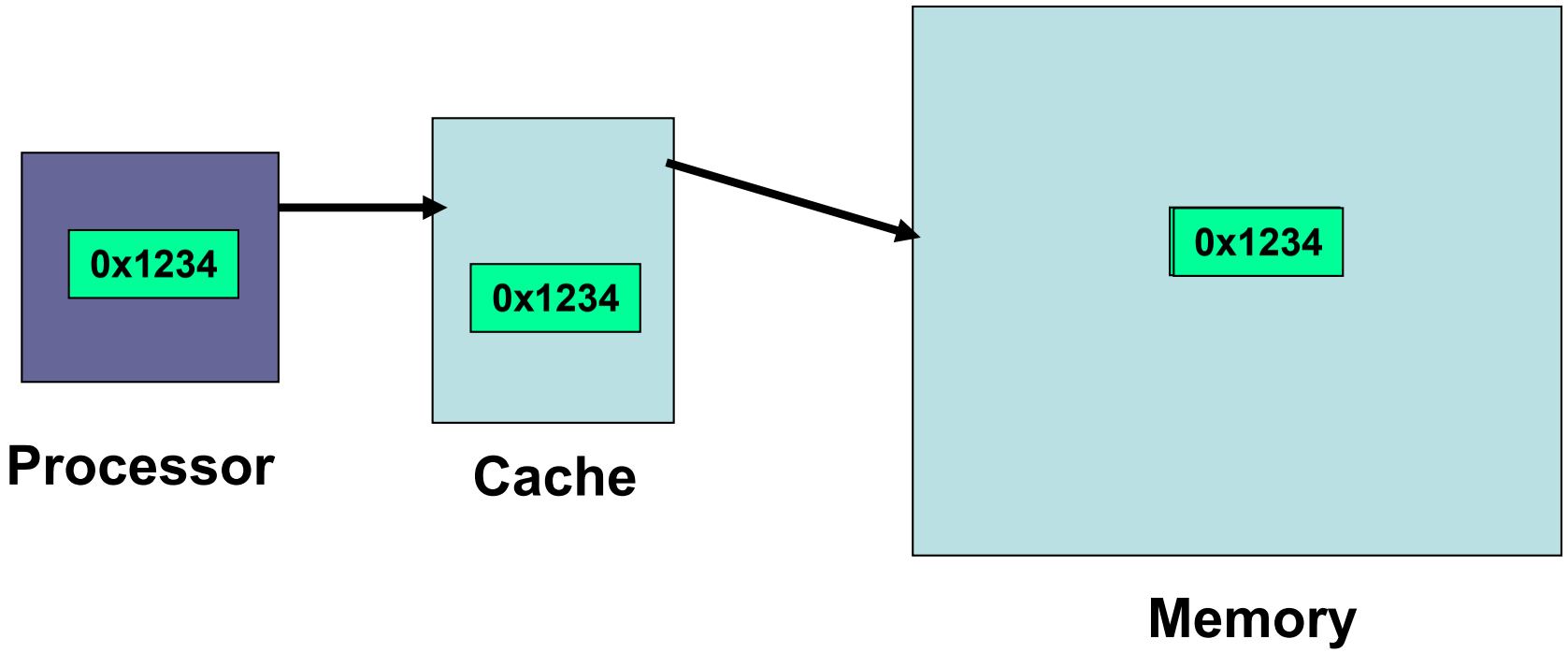
The value is written to both the cache line and to the lower-level memory (e.g. main memory).

Write Buffer

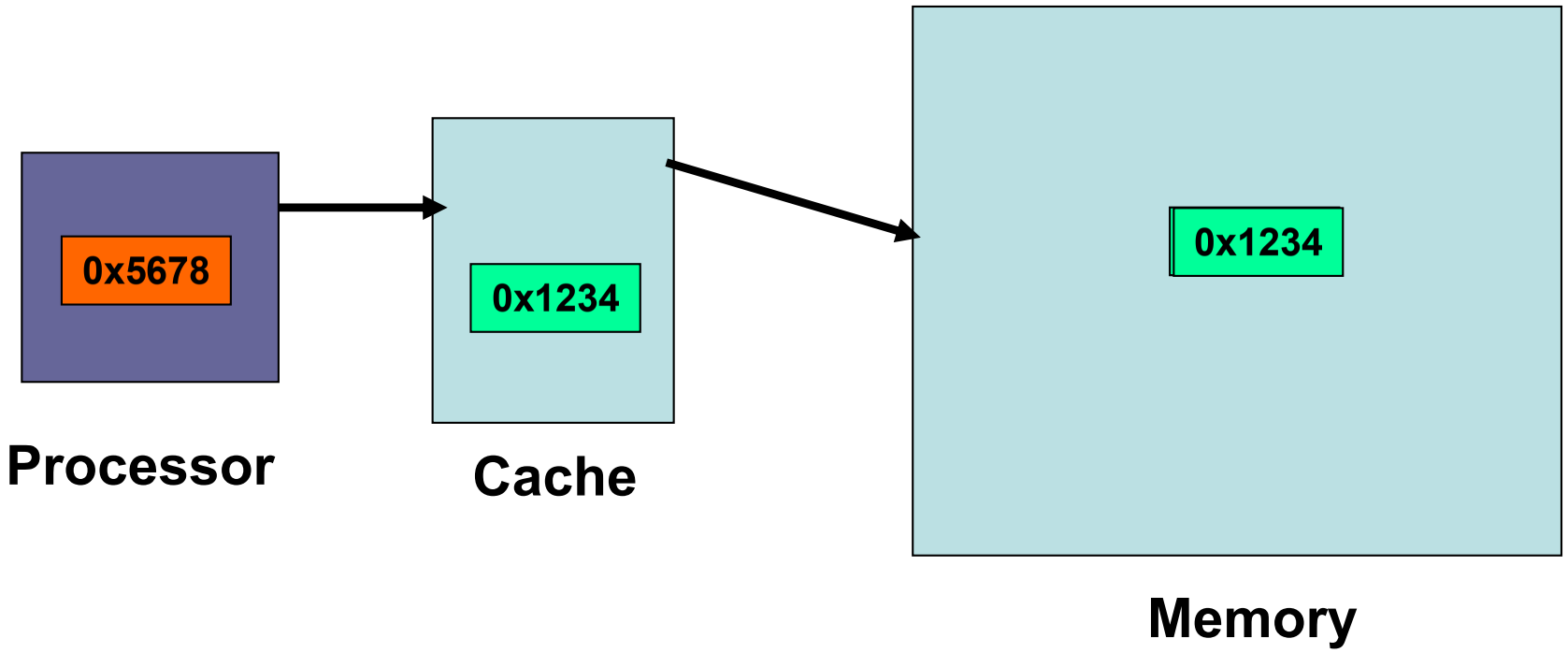


- Processor writes data into the cache and the write buffer
- Memory controller writes contents of the buffer to memory
- Write buffer is a FIFO structure:
  - Typically 4 to 8 entries
  - Desirable: Occurrence of Writes << DRAM write cycles

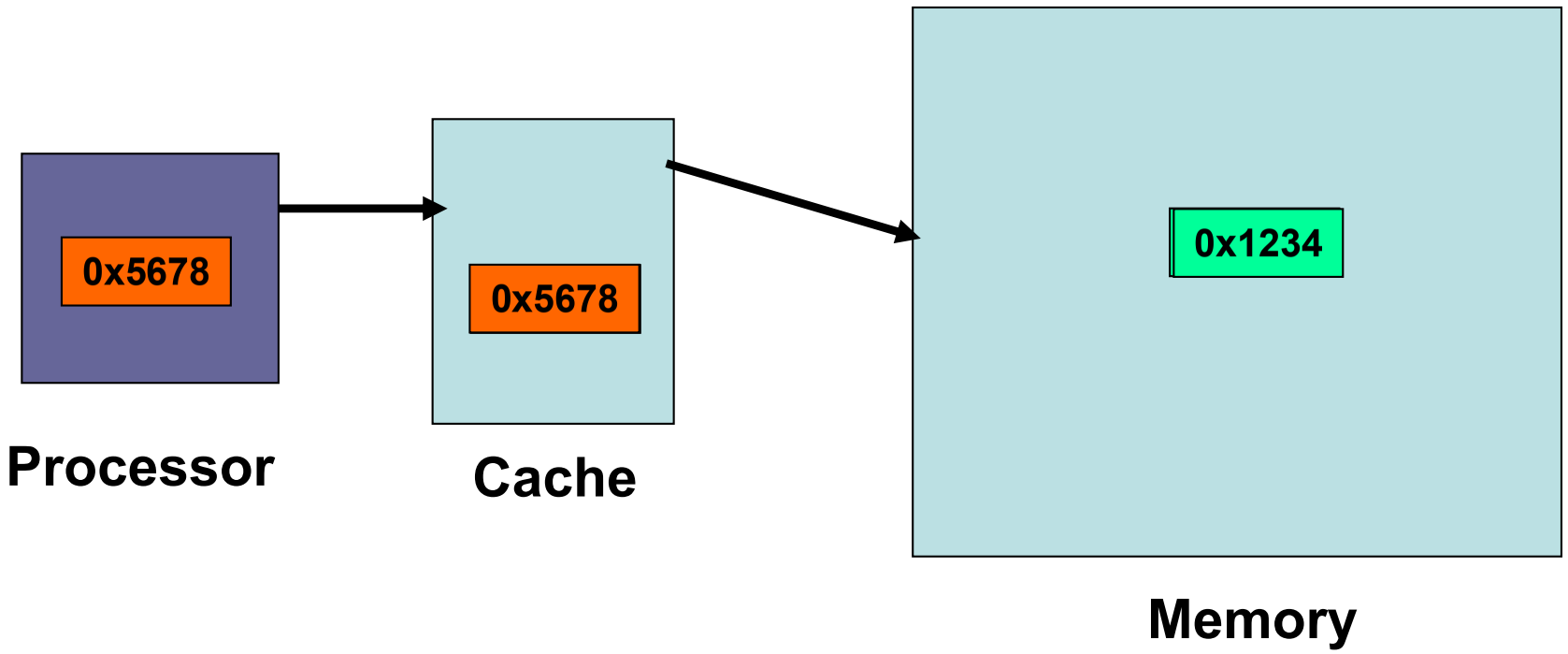
Writeback Policy



Writeback Policy

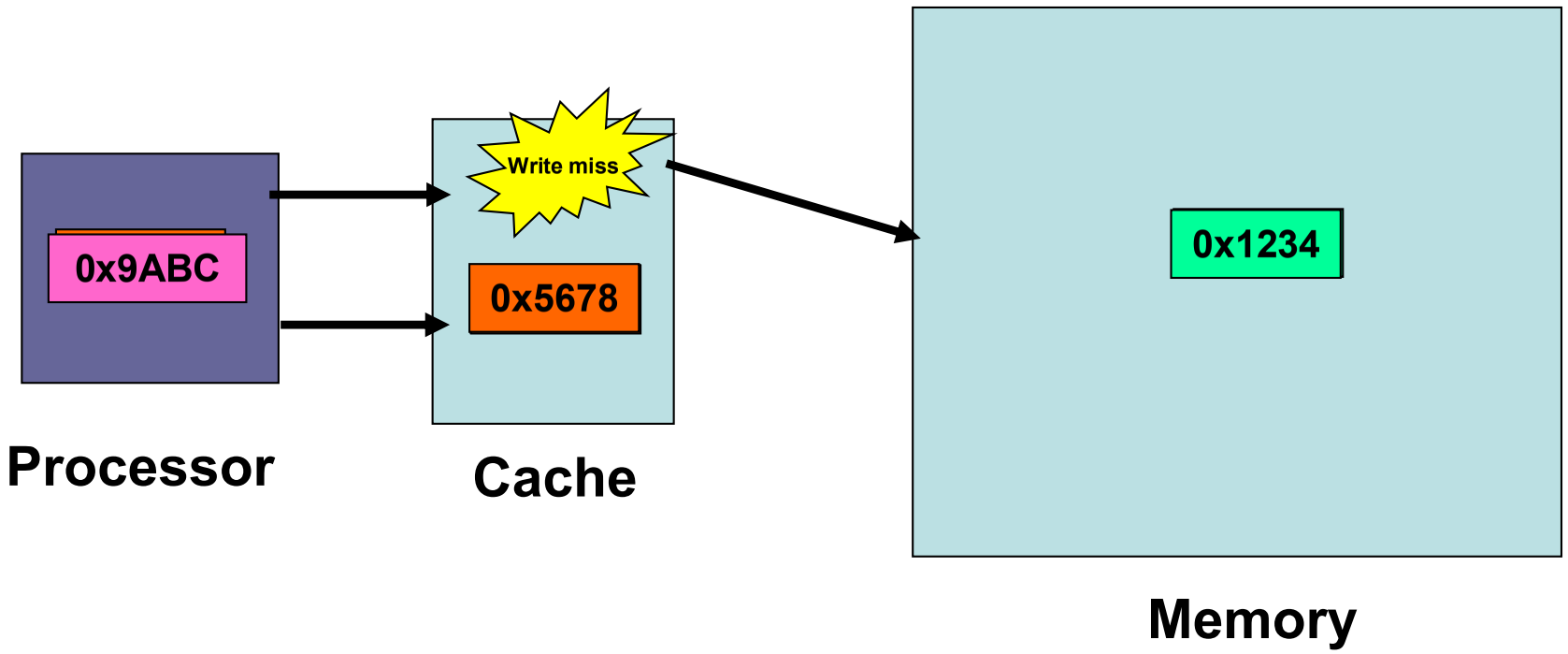


Writeback Policy



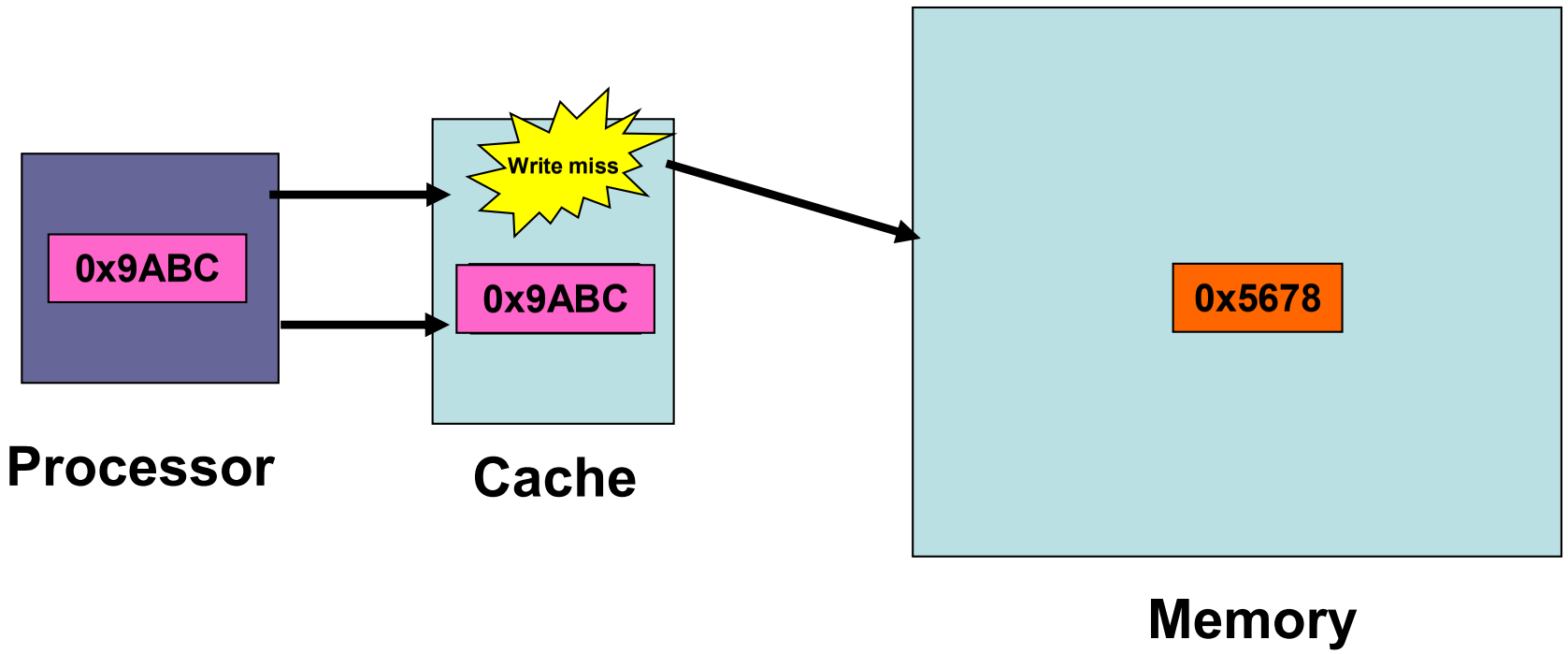
The value is written only to the cache line. The modified cache line is written to main memory only when it has to be replaced.

Writeback Policy



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.

Writeback Policy



The modified cache line is written to main memory only when it has to be replaced.

What happens if Write Miss?

- *Write allocate*
  - After write miss, the data will be loaded to cache first before write
  - 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

Topics covered

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

Supplementary Reading Materials

Computer Organization and Design: The Hardware/Software Interface

Fifth Edition

Chapter 5:
Large and Fast: Exploiting Memory Hierarchy

Section 5.1 - 5.4