

# Fundamentals of MCS Computer Architecture #2

Toshio Endo

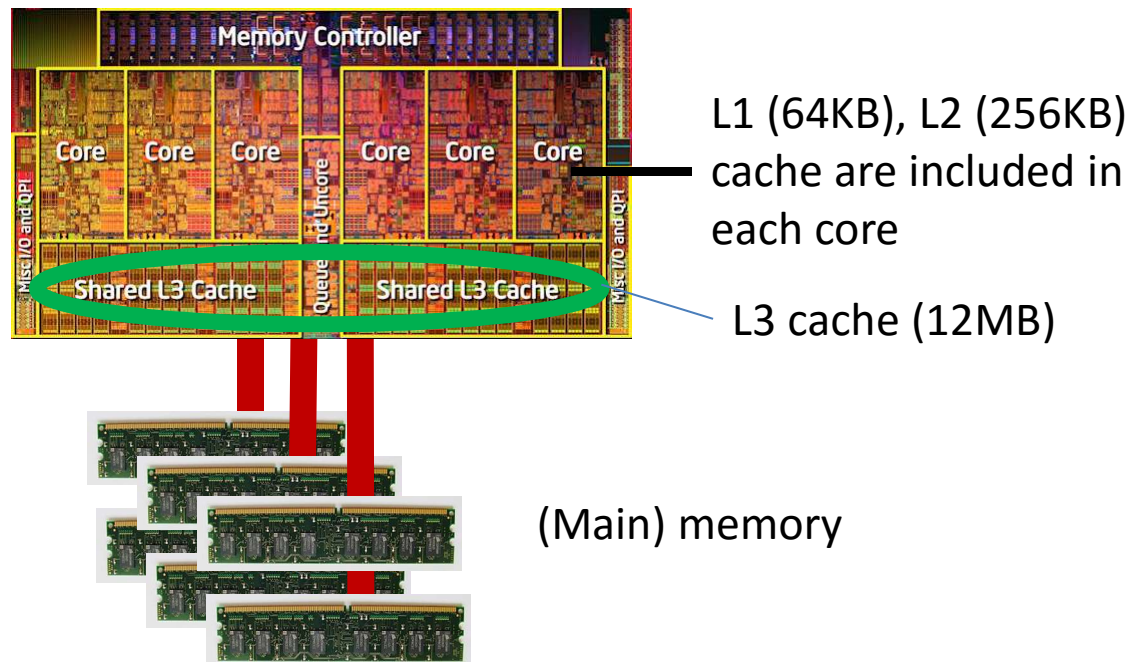
[endo@is.titech.ac.jp](mailto:endo@is.titech.ac.jp)

[www.el.gsic.titech.ac.jp](http://www.el.gsic.titech.ac.jp)

## **CACHE MEMORY (CONT'D)**

# Cache Memory

- Fast and small memory (usually) included in CPU
- Used to store data that have been **recently accessed**
- Used automatically --- Sometimes programmers do not know existence of cache memory



# Memory Access with Cache

Example: CPU executes “read access to address 0x12345642”

1. Calculate the start address of cache line that includes target address
2. Search address 0x12345640 in cache
  - 2-1: If found, **cache hit** (We go to Step 5.)
  - 2-2: If not found, **cache miss** (This is the case now)
3. Select a “victim” line in cache, to be deleted
4. Copy 64byte data from [0x12345640, 0x1234567F] in memory to cache (This takes >100 clocks)
5. Deliver the desired data to CPU core

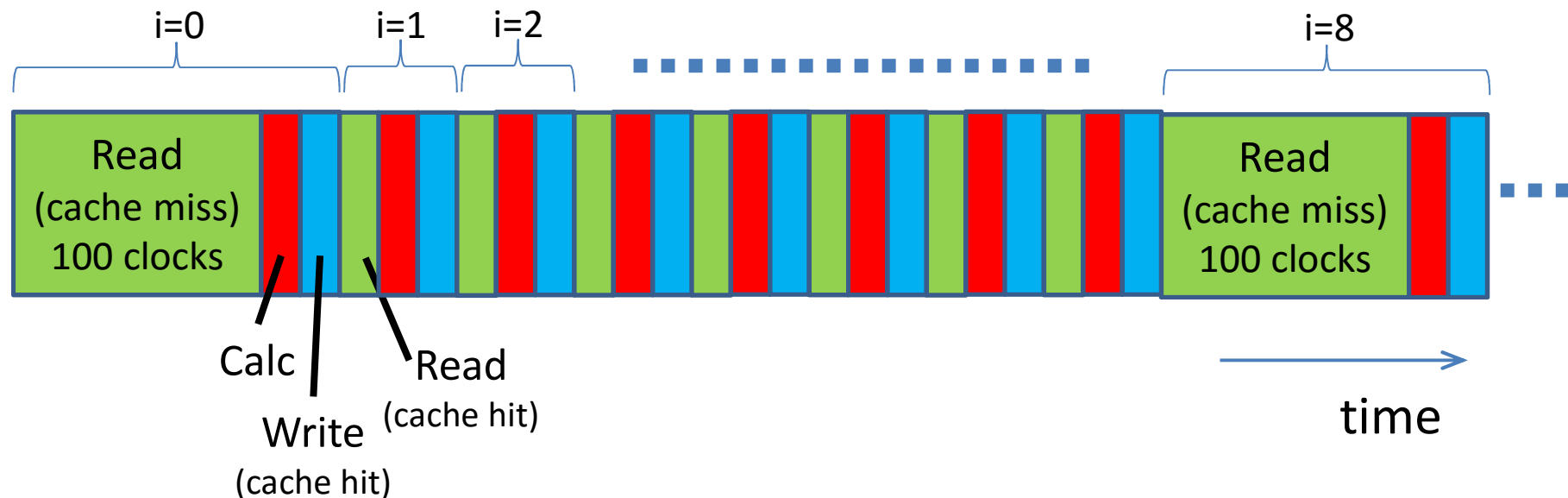
# Characteristics of CPU with Cache

- Time to execute a memory access instruction is not constant
  - In cache hit cases, a few clocks
  - In cache miss cases, >100 clocks
- Due to existence of cache lines, sequential memory access tends to raise higher cache hit ratio
  - Program A accesses to 12345642, 12345644, 12345646...  
→ Good locality
  - Program B accesses to 12345642, 1234A000, 23456780...  
→ Bad locality

# Example of Sequential Access

```
for (i = 0; i < n; i++) A[i] = A[i]*2.0;
```

- We assume that cache is empty, when the programs begins
- We assume A[i] has double type (8Byte)



- Much more efficient than “No cache” CPU in p. 4
- Actual CPU is even more efficient, due to pipelined execution

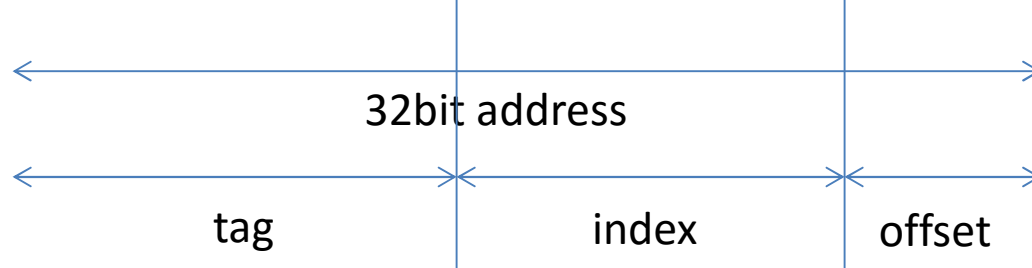
# Deeper Insights: Cache Design Policy

- How the “victim” line is selected?
    - Direct mapping, or set associative or full associative
  - “Write” is more complex than read!
    - Write through, or Write back
- These policy is chosen by processor makers (Intel, AMD, Intel...), so users cannot change it
  - Memory access is done by hardware (not software), too complex method is impractical
    - For example, there is no commercial CPU with full associative cache

# How the “victim” line is selected?

## Direct Mapping Method (1)

- Let the target address be 0x12345642  
= 00010010001101000101011001000010 (binary notation)



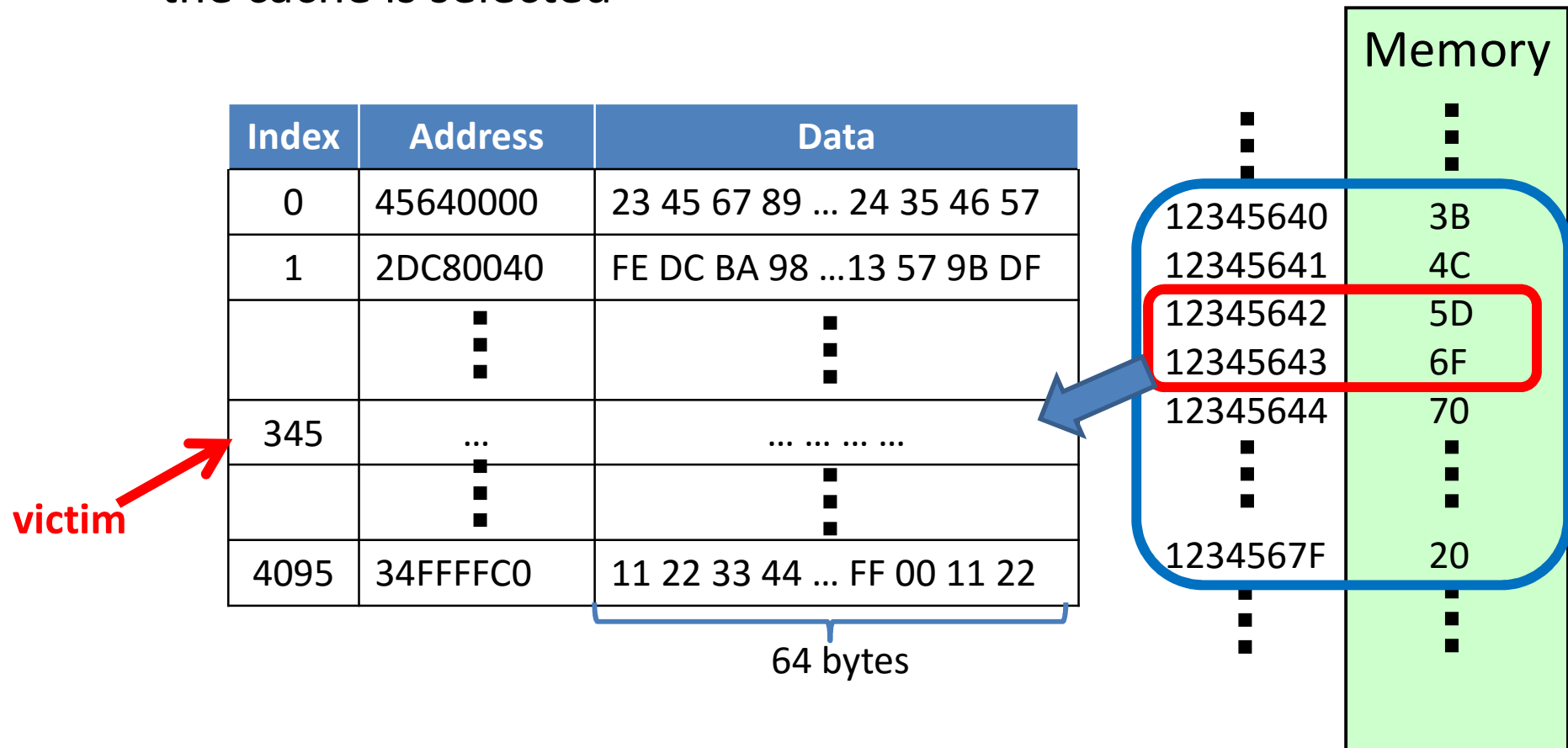
- In our example cache (64B x 4K lines = 256KB),
  - offset is 6 bit (=log 64)
  - index is 12 bit (= log 4096)
  - tag is 14 bit (= 32-12-6)



# How the “victim” line is selected?

## Direct Mapping Method (2)

- Victim is determined by *index* of the address
- When index is  $000101011001 = 0x159 = 345$ , 345-th line in the cache is selected



# Problems of Direct Mapping

- Direct mapping looks like a simple “hash” algorithm
- In *unlucky* cases, more cache misses may occur than expected

```
for (i = 0; i < n; i++) A[i] = A[i]+B[i];
```

- If address A and address B has the same index (unfortunately), cache hit ratio = 0%
- Full Associative method with LRU policy could avoid hash collision, but impractical
- Remember that cache is made of hardware/circuit!
- An intermediate method is called **set associative method**

# How the “victim” line is selected?

## *n*-way Set Associative Method (1)

- Cache memory hold *n* tables of cache lines
  - *n*=1 means direct mapping
  - The figure shows *n*=2 (**2-way set associative**) case
    - Typically, *n*=4 ~ 16


cache							
0-way Table				1-way Table			
Index	Address	Data	FI	Index	Address	Data	FI
0	45640000	... ..		0	C39C0000	... ..	X
1	2DC80040	... ..	X	1	15280040	... ..	
:	:	:		:	:	:	
345	...	... ..	X	345	...	... ..	
:	:	:		:	:	:	
2047	34FFFFC0	... ..		2047	687FFFC0	... ..	X

# How the “victim” line is selected?

## *n*-way Set Associative Method (2)

- When index is obtained (345 for example), there are *n* **candidates** for victim
- The basic policy is “to keep data recently accessed”
- If  $n=2$ , a flag is used to show recently accessed line

Index	Address	Data	FI	Index	Address	Data	FI
0	45640000	... ..		0	C39C0000	... ..	X
1	2DC80040	... ..	X	1	15280040	... ..	
:	:	:		:	:	:	
345	...	... ..	X	345	...	... ..	
:	:	:		:	:	:	
2047	34FFFC0	... ..		2047	687FFFC0	... ..	X

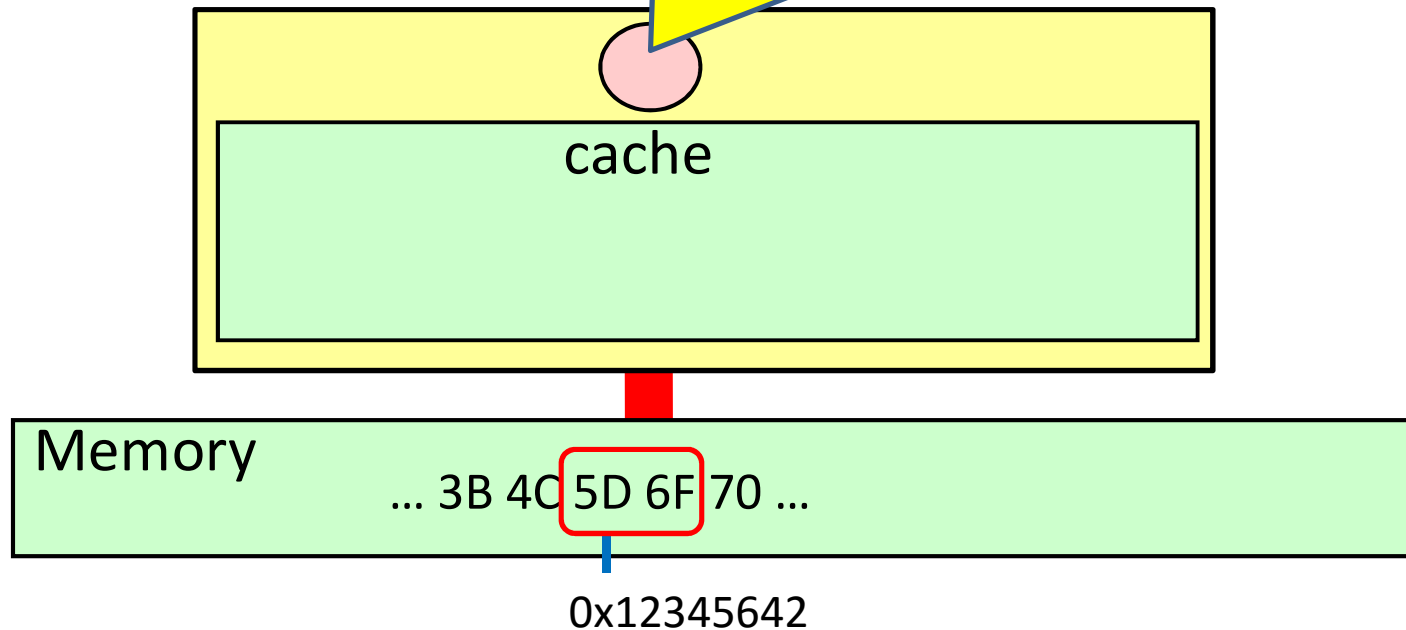
**victim** 

If  $n>2$ , more complex mechanism is used such as Quasi LRU

# Write (1)

- “Write” is different from “Read” since it changes data

“write 0xAB 0xCD (2byte) to 0x12345642”

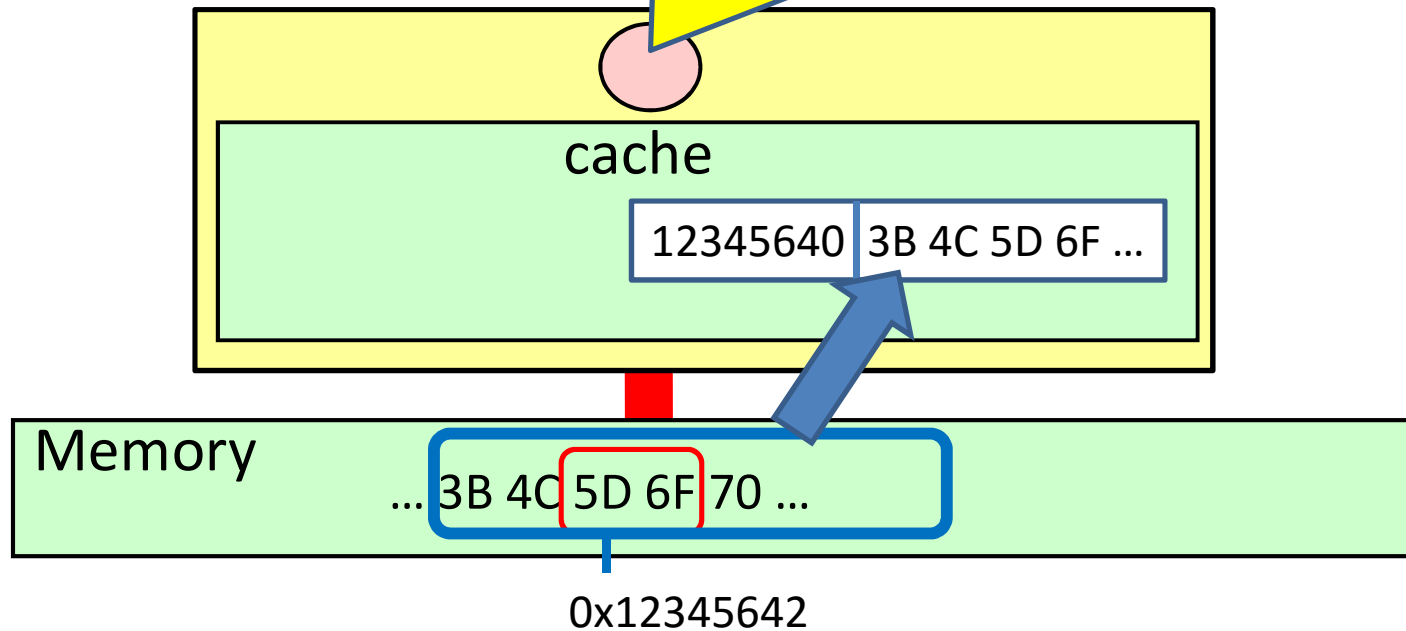


# Write (2)

Step 1. 2. 3. are same as “Read”

4. Copy 64Byte data (cache line) to cache

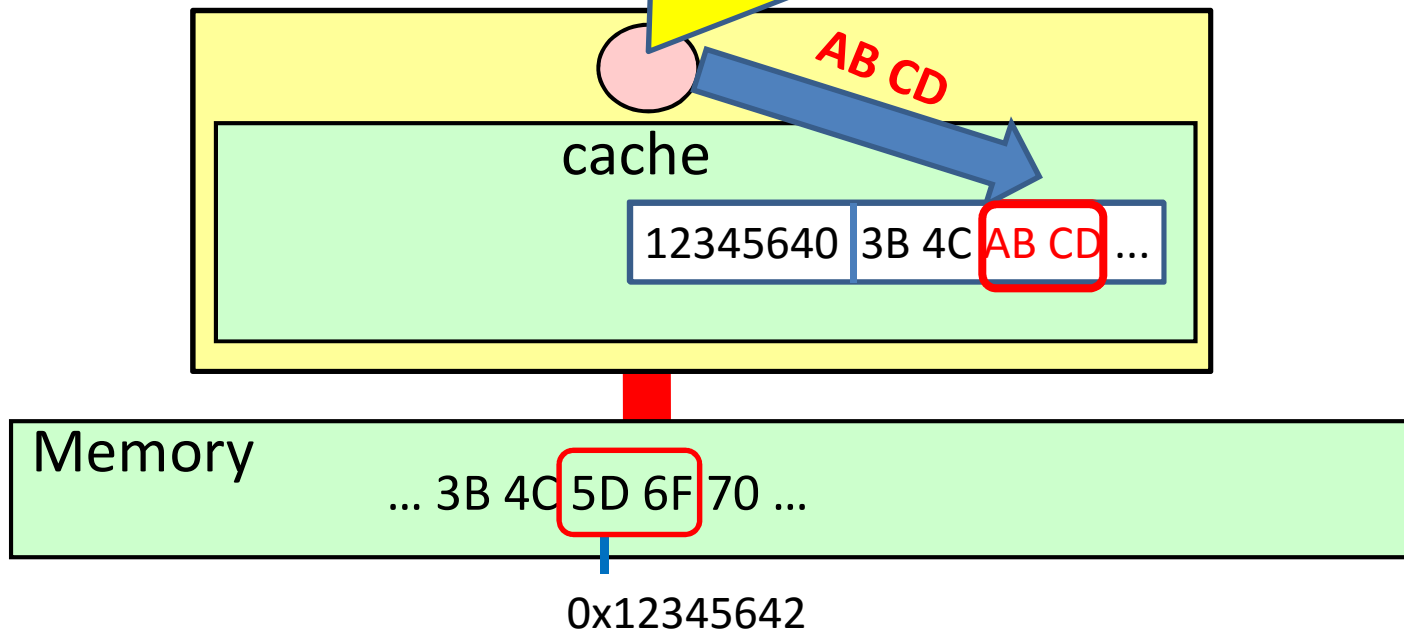
“write 0xAB 0xCD (2byte) to 0x12345642”



# Write (3)

## 5. Update data in cache

“write 0xAB 0xCD (2byte) to 0x12345642”



Memory has not updated yet. How should we do?  
Two policies: Write through , or Write back

# Write Policies

- Write through
  - Update data on memory immediately
  - **Problem**: Every write instruction takes >100 clocks
- Write back (**more popular**)
  - Data on memory is **updated later**
  - **When** the line is to be **deleted as a victim**, due to future cache misses
  - Hardware becomes more complex, but efficient



Due to this time lag, multi-core cache becomes complex


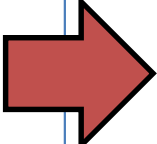

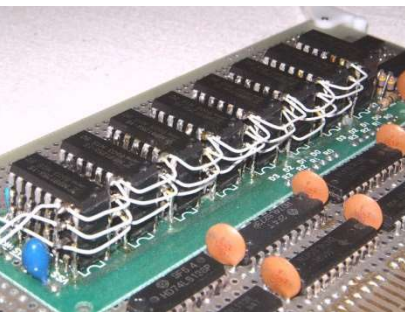
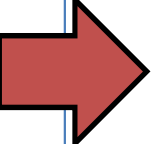



# Summary for (sequential) Cache Memory

- Memory performance is highly ( $\times 20 \sim 100$ ) affected by existence of cache memory
- Considering cache line is important
  - Spatial locality
  - 64 Byte in current Intel CPUs

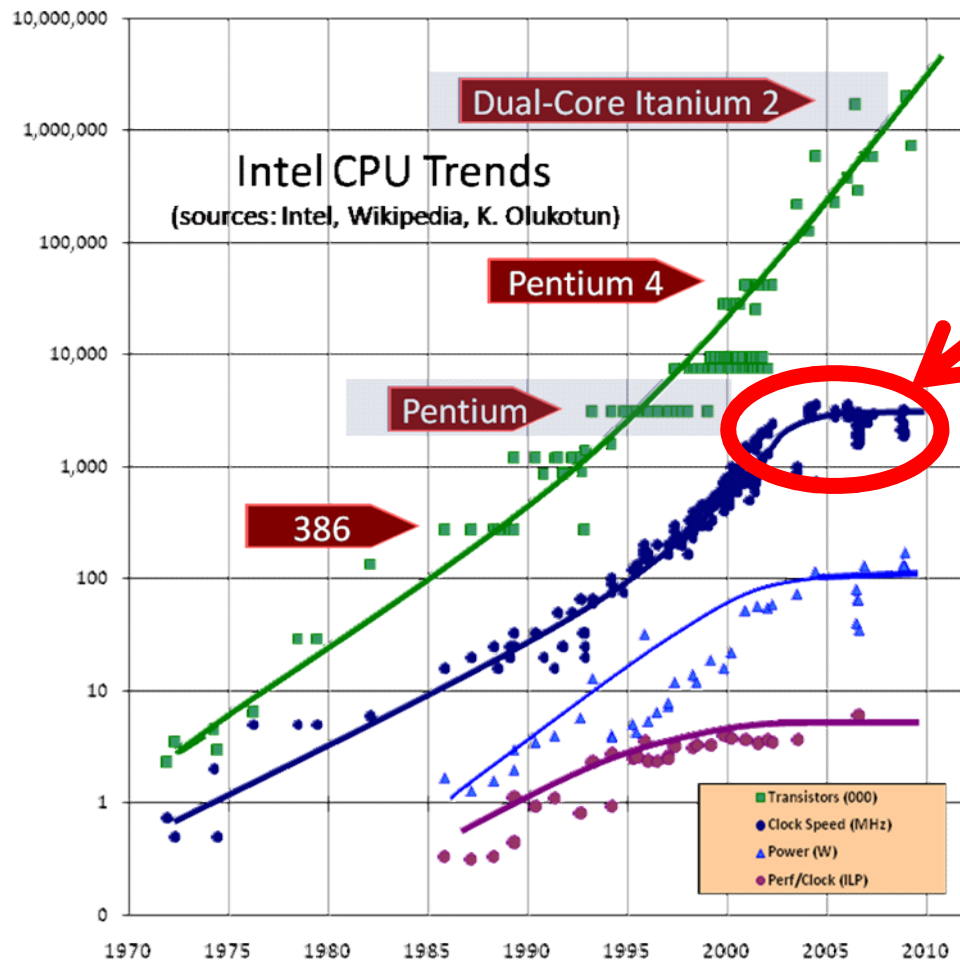
# PARALLELISM

# Improvement of Processor Clock

	Around 1980		Present
CPU	2MHz → 1clock = 500ns 	x1000 	2GHz → 1clock = 0.5ns 
Memory	Access time = 2000ns(?) 	x40(?) 	Access time = 50ns or more 

Does processor clock further get faster?  
Will we have 2THz CPUs 30years later?

# Trend of Processors

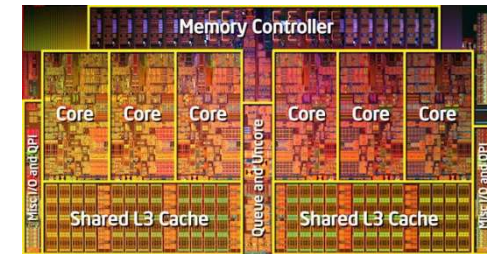


- Growth of clock speed of processors has stopped around 2003
- Instead, the number of cores is increasing
  - Even cellphones have dual-core processors

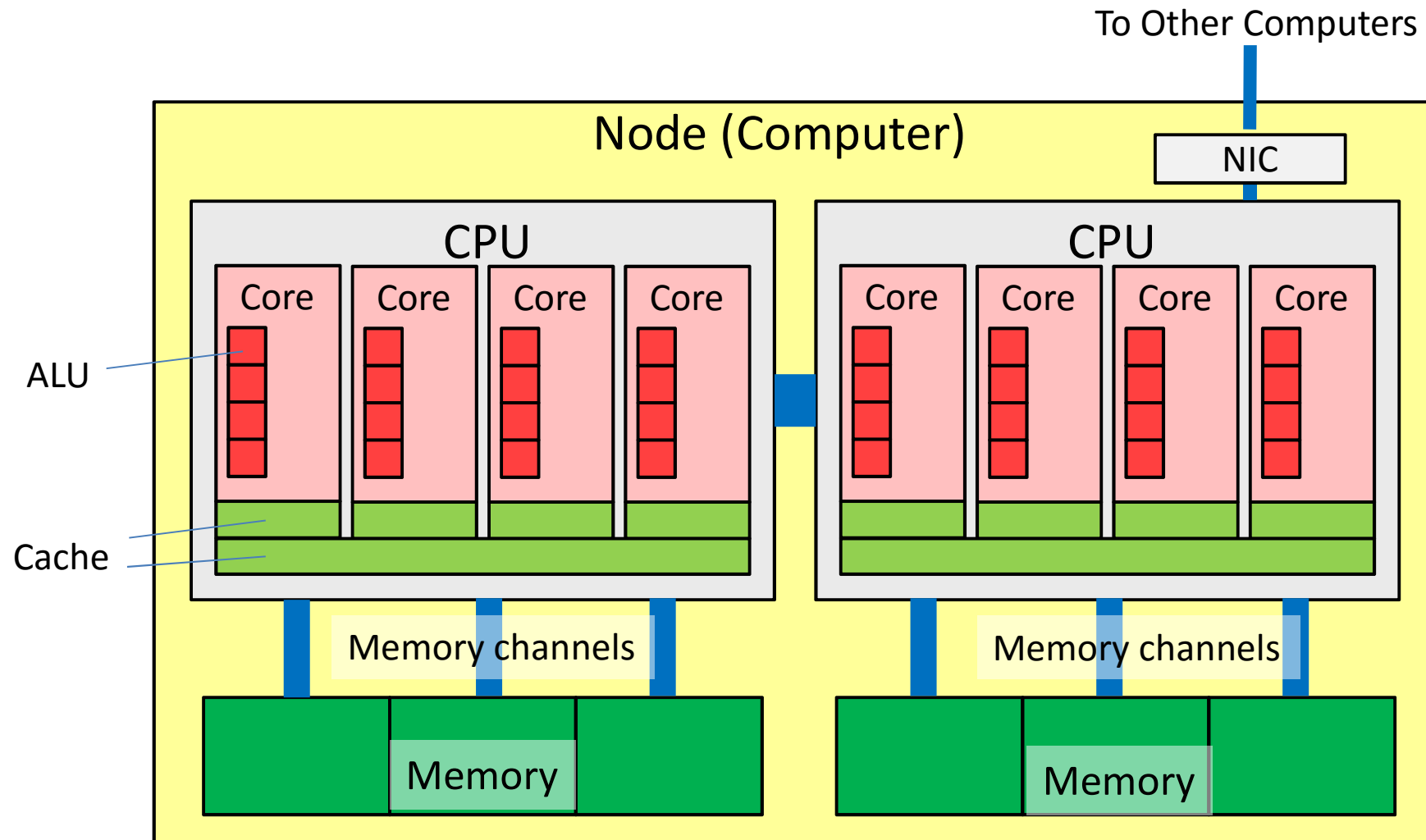
“Slow&Parallel” approach

# Hierarchy of Parallelism

- SIMD parallelism
  - multiple operations can be done simultaneously (SIMD)
  - MMX, SSE, AVX
- Multi-Core (Multi-CPU) parallelism
  - Multiple cores can work cooperatively
  - Pthread, Java thread, OpenMP
- Using GPU (skipped in this lecture)
  - Thousands of GPU cores
  - CUDA, OpenCL, OpenACC
- Multi-Node parallelism
  - Multiple nodes can work cooperatively
  - Socket, Hadoop, MPI

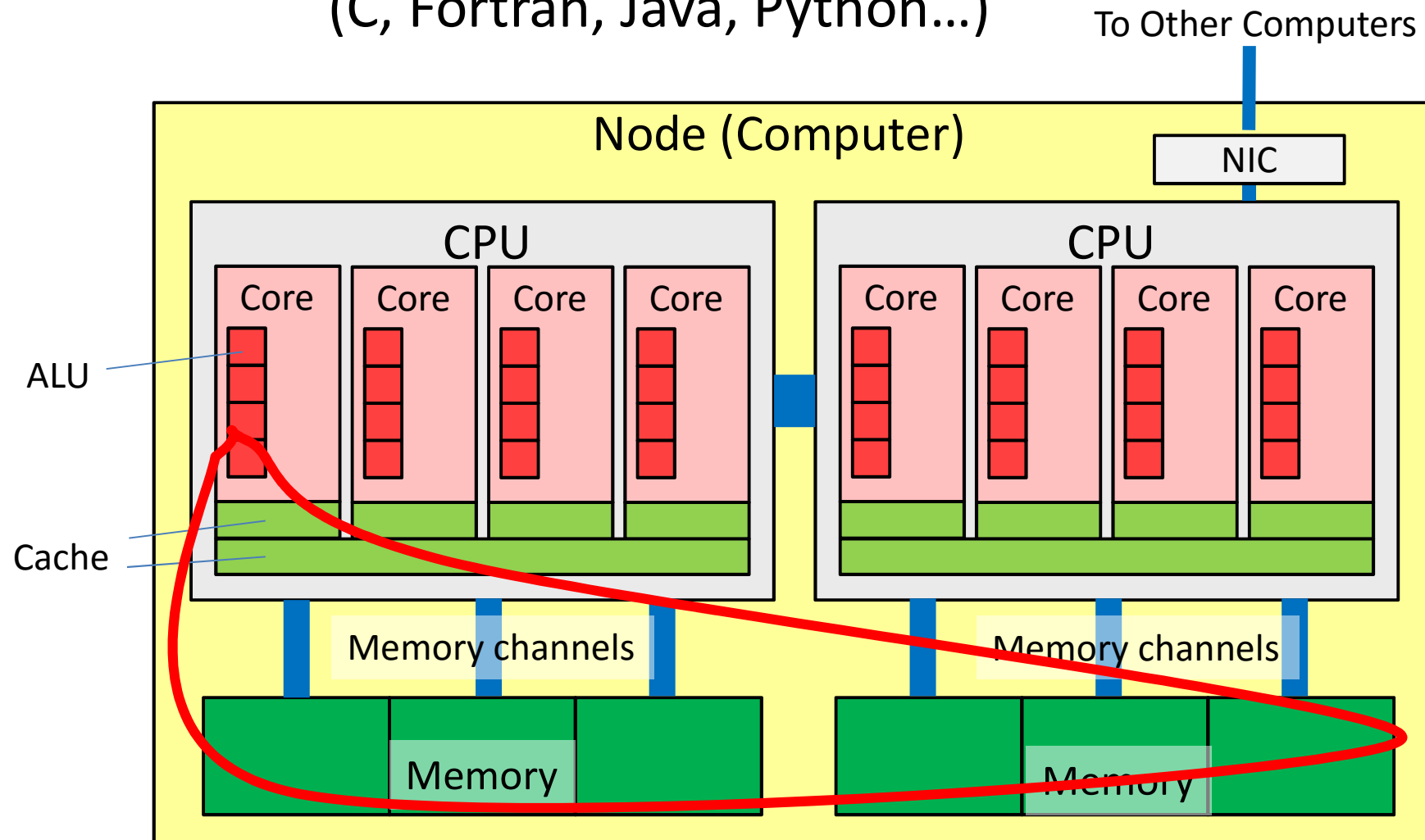


# Hierarchy of Parallelism in Modern Computers

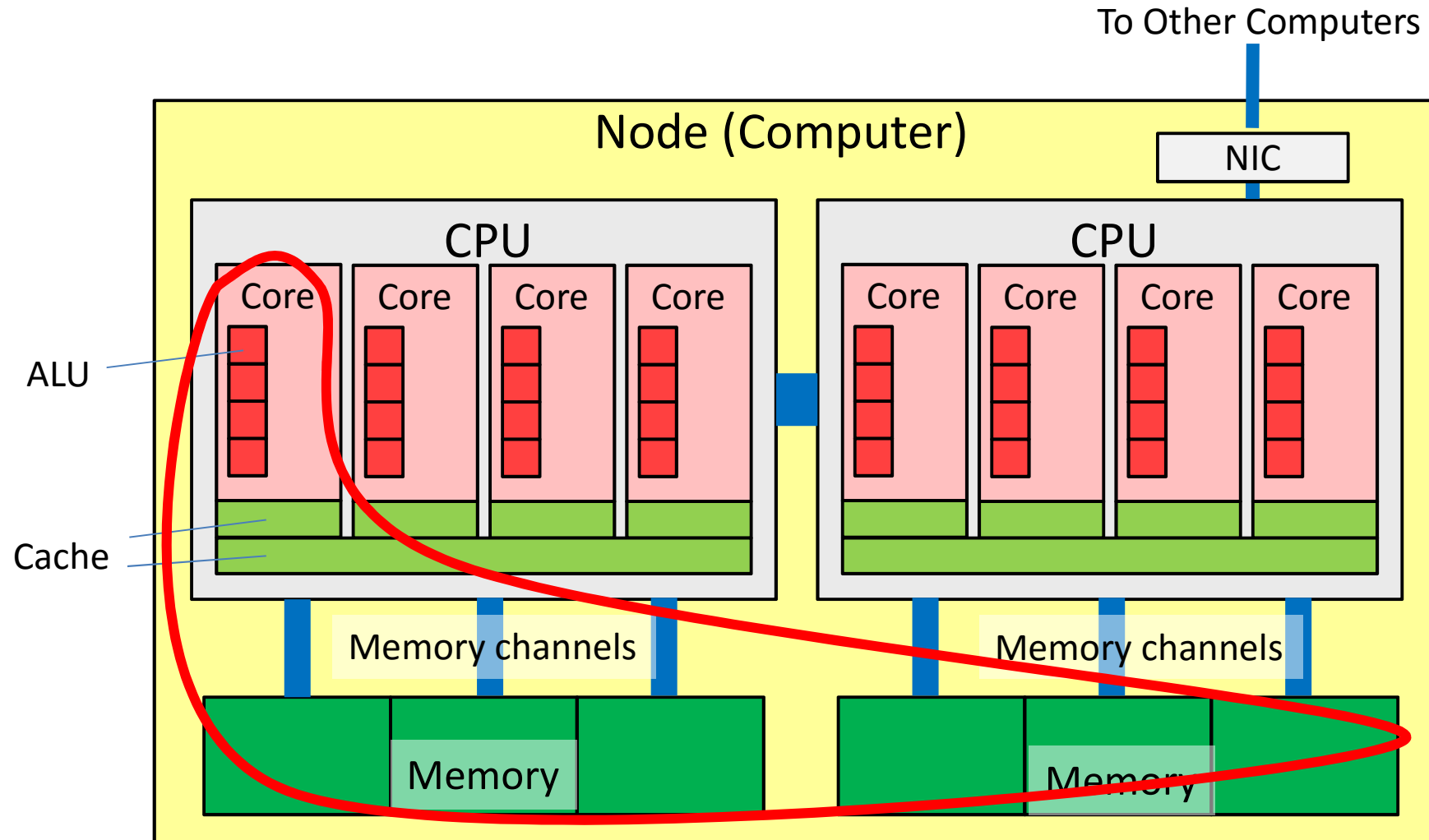


# No Parallelism

## With Typical Programming Languages (C, Fortran, Java, Python...)

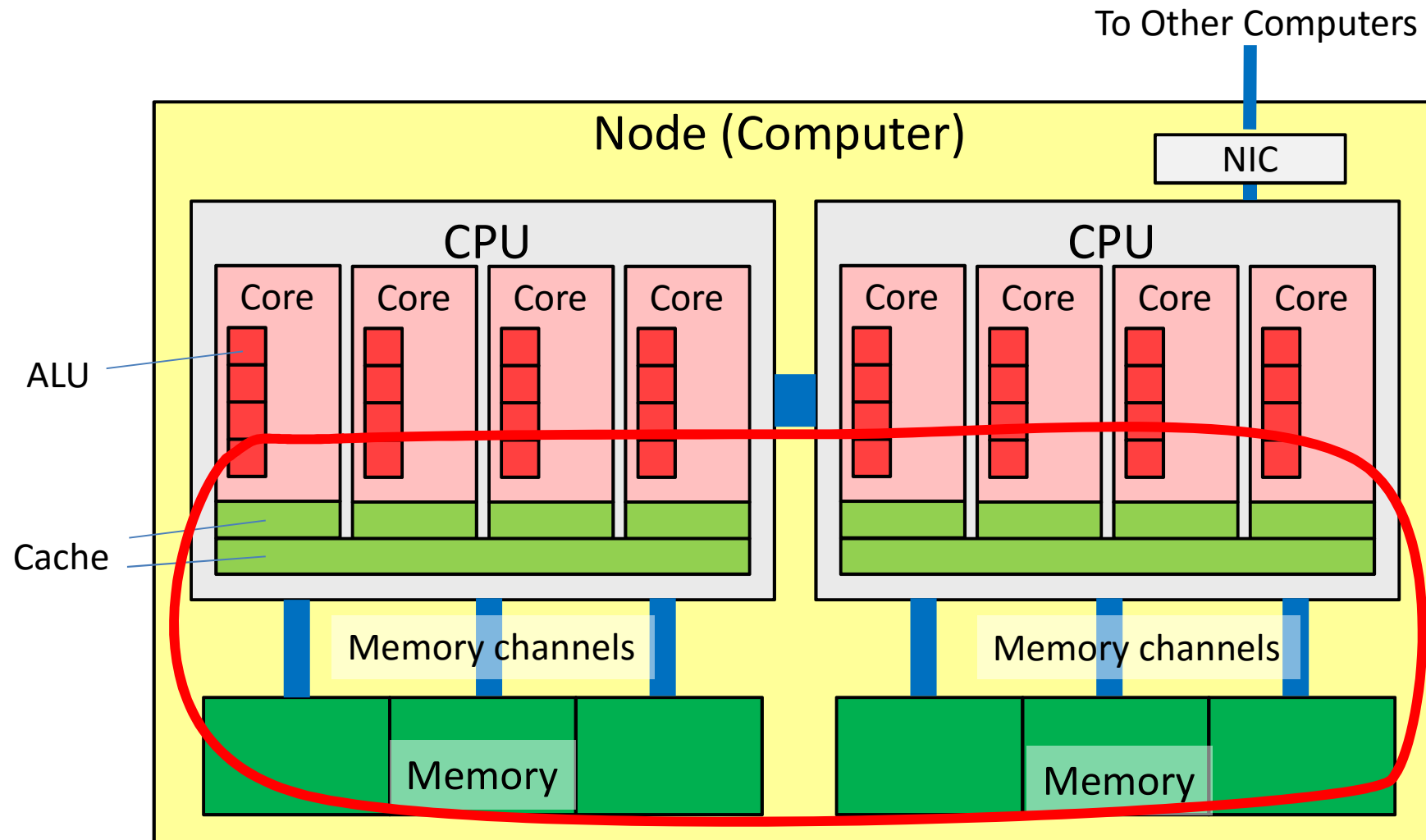


# SIMD parallelism (SSE, AVX...)



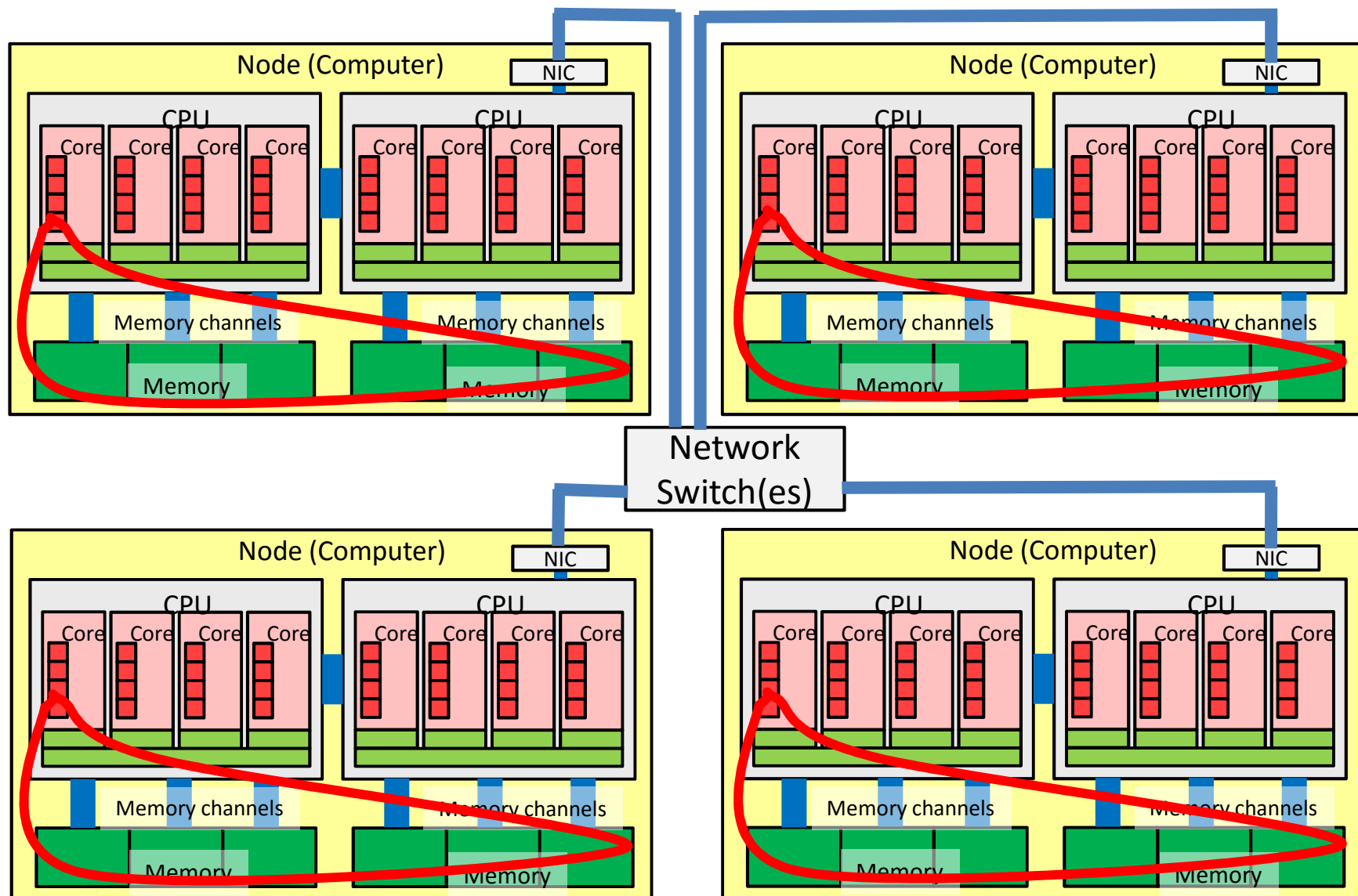


# Multi-Core/Multi-CPU Parallelism (Pthreads, Java threads, OpenMP...)



# Multi-Node Parallelism With Communication

(Sockets, Hadoop, MPI...)



# Next Lecture

- Jan 18 (Mon) is cancelled
- Jan 25 (Mon): Parallelism (Cont'd)
  - SIMD, Multi-core
  - Maintaining consistency of cache
  - About the report of this part