

Chapter 3

Memory Hierarchy



Memory

- Register
- Cache
- Memory
- Storage
- Mechanical memory
 - Acoustic wave/torque wave delay line memory
 - Magnetic Drum Memory
 - Magnetic core memory
- Electronic memory
 - SRAM
 - DRAM
 - SDRAM
 - Flash
 - ROM
 - PROM
 - EPROM
- Optical memory

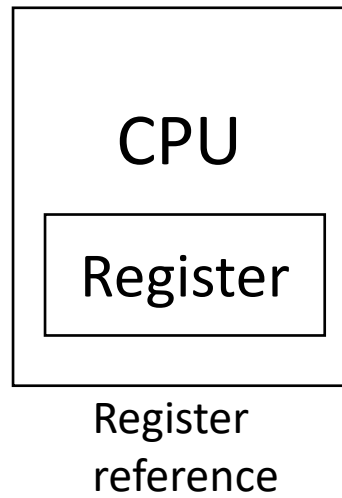


Memory?

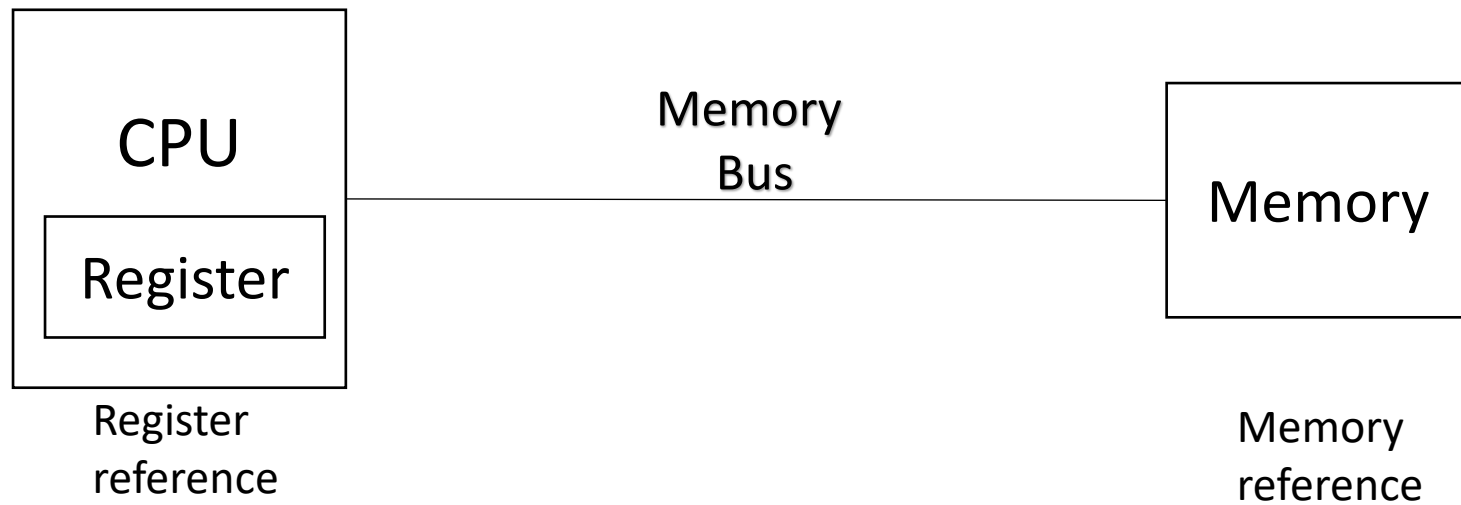
- Load $R2, 0(R1)$
- Store $R2, 0(R1)$



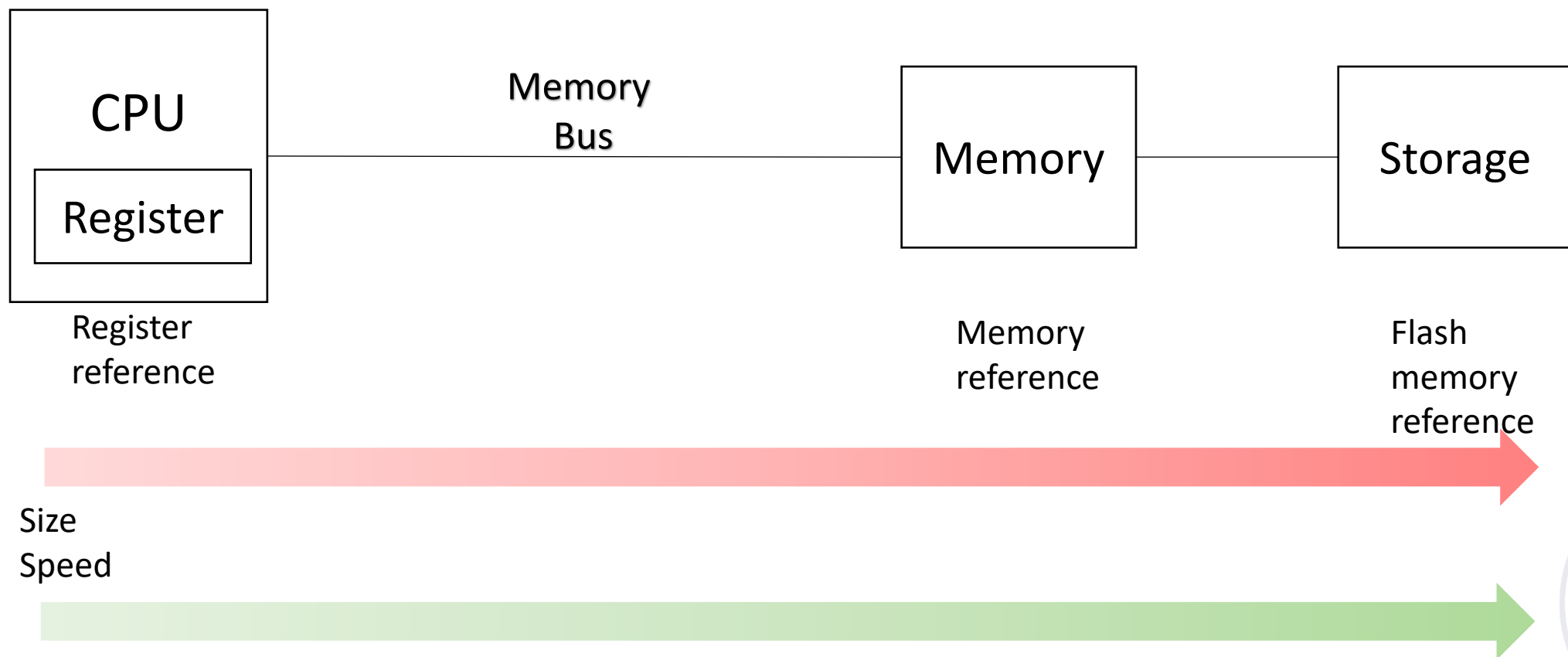
Data processing & temporary storage



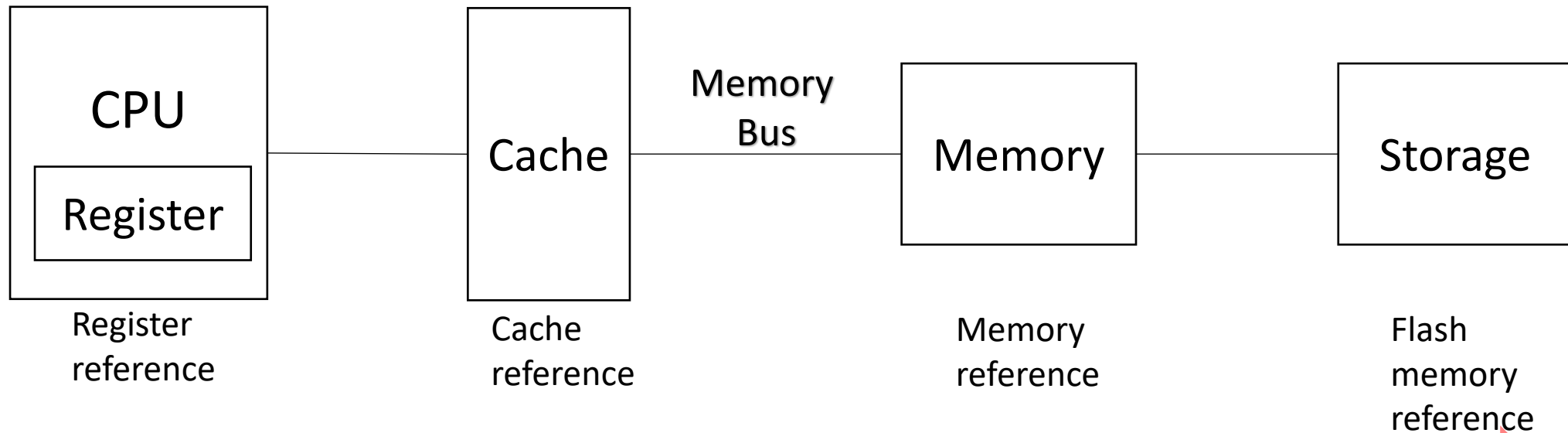
Temporary storage



Faster temporary storage



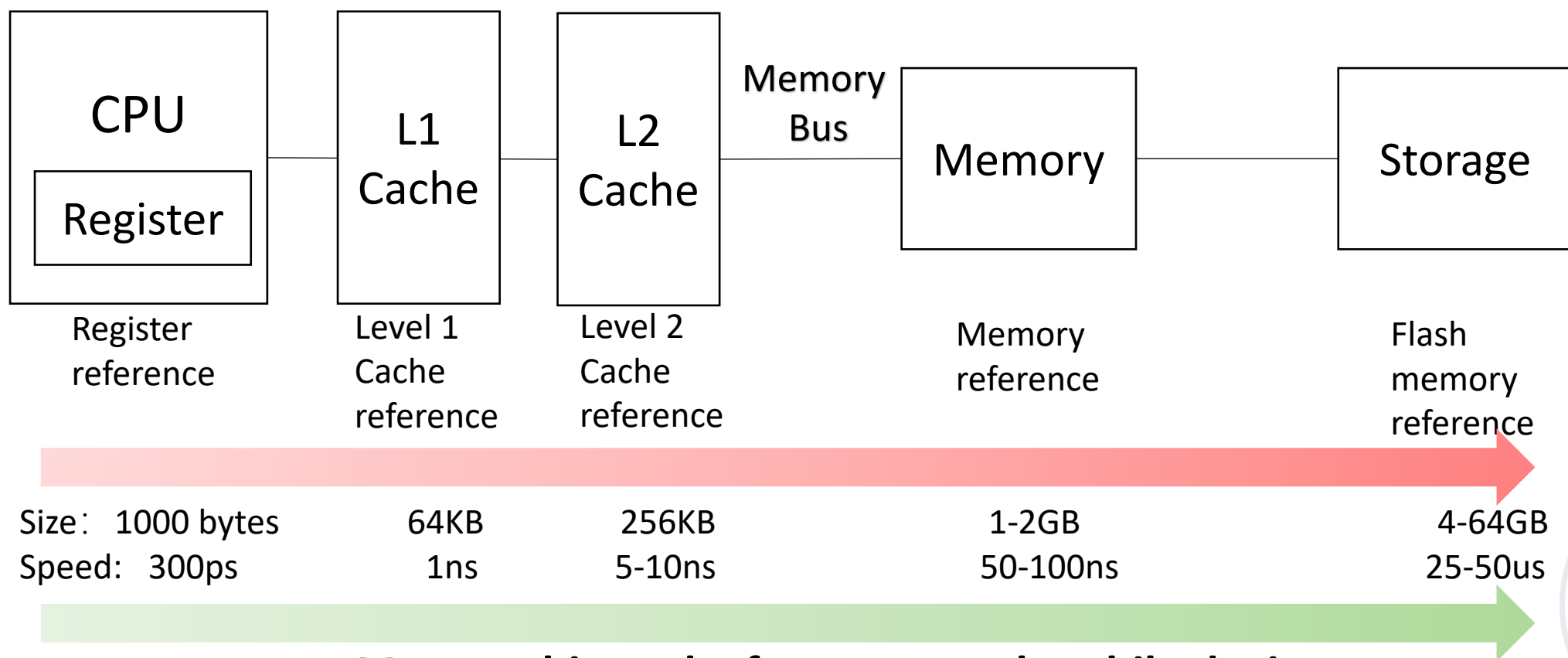
Faster temporary storage



Size
Speed



Memory Hierarchy

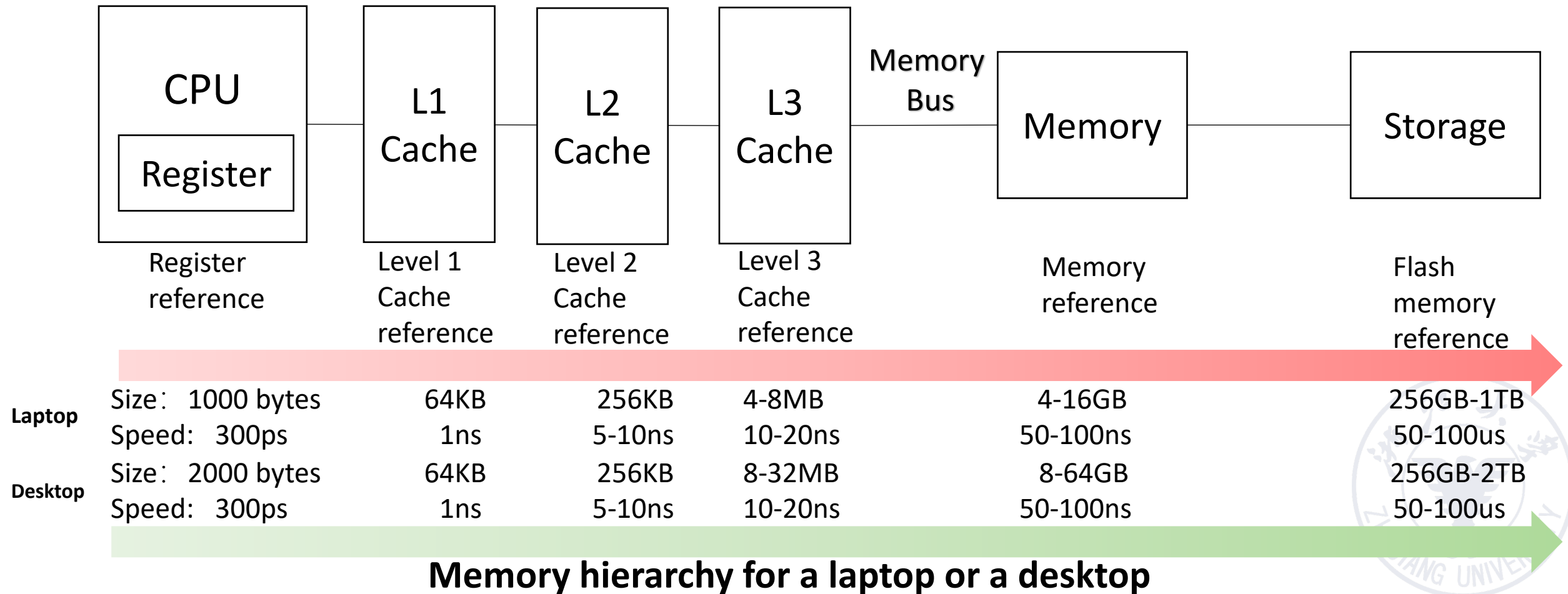


Memory hierarchy for a personal mobile device

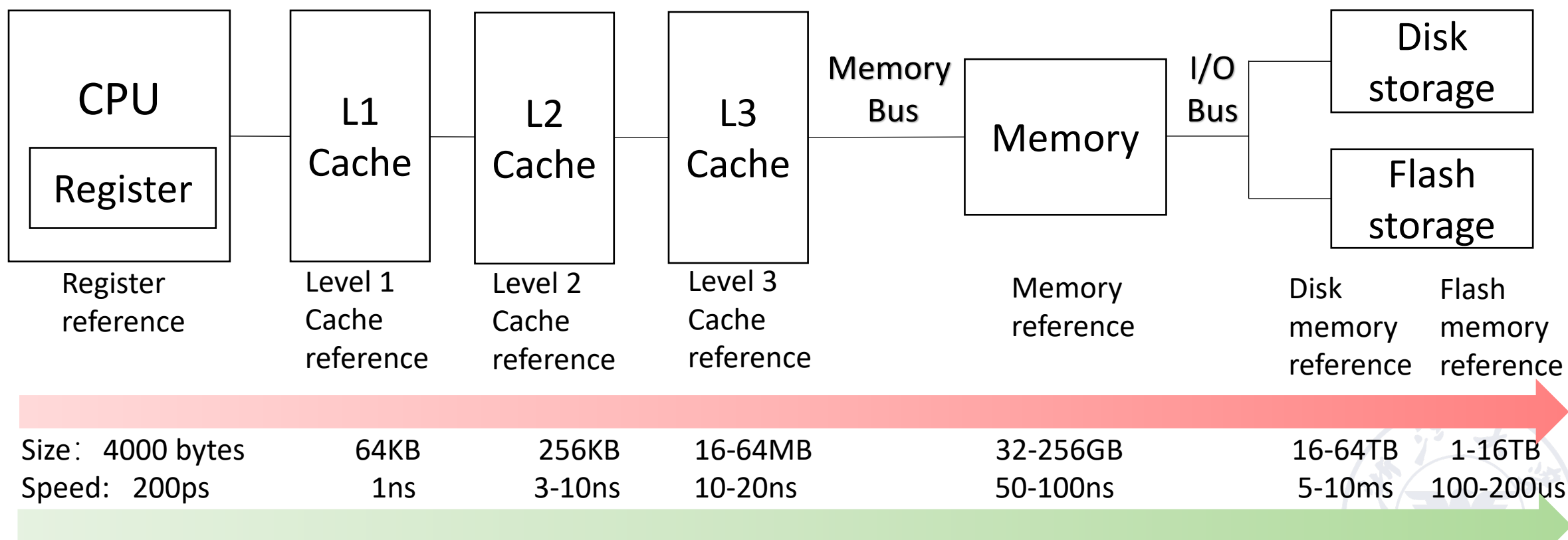
*1000 picoseconds = 1 nanosecond = 10^{-6} millisecond



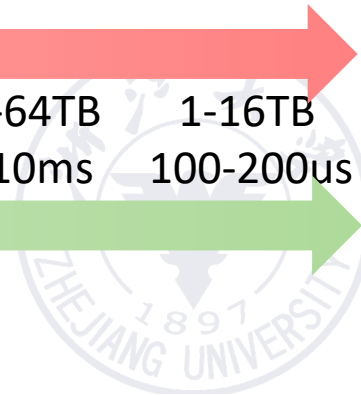
Memory Hierarchy



Memory Hierarchy



Memory hierarchy for server



Wait, but what's cache?



Wait, but what's **cache**?

data request?



**program/
instructions**

in / out?



Wait, but what's **cache**?

data request?



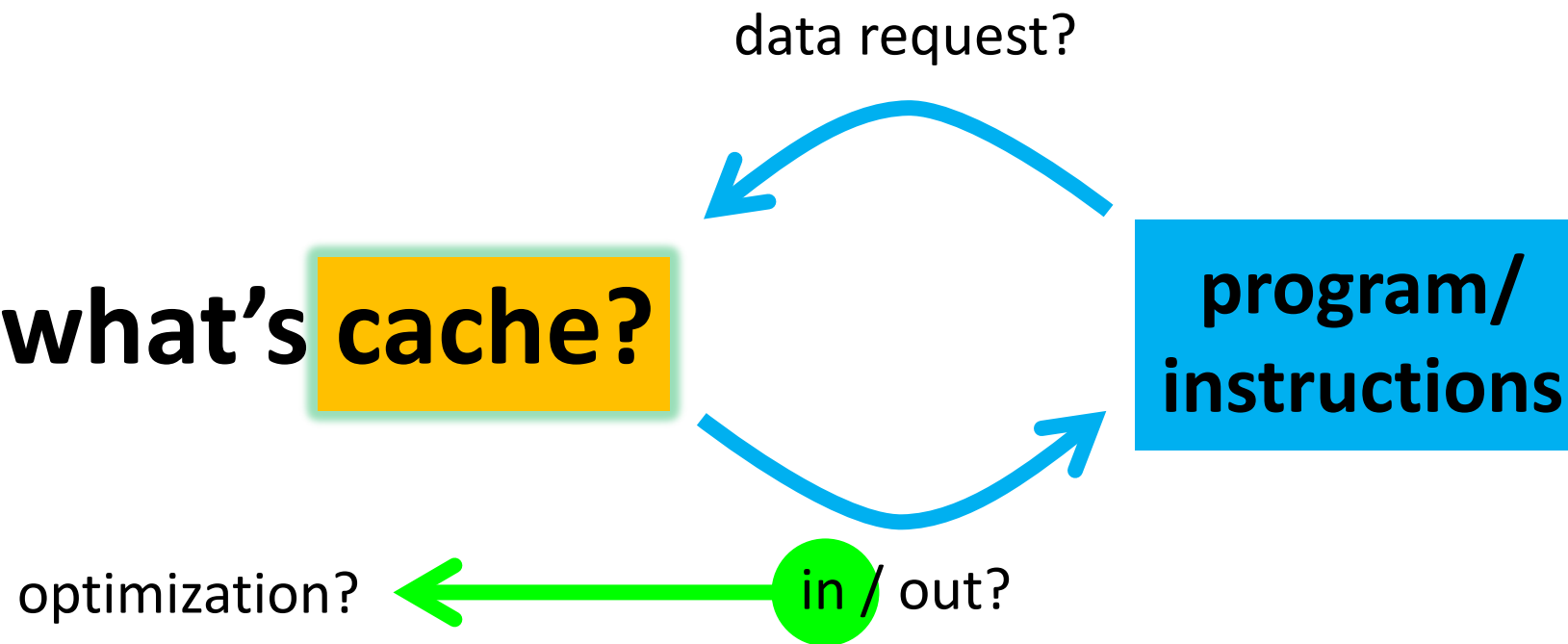
**program/
instructions**



in / out?



Wait, but what's **cache**?



So, what's cache?



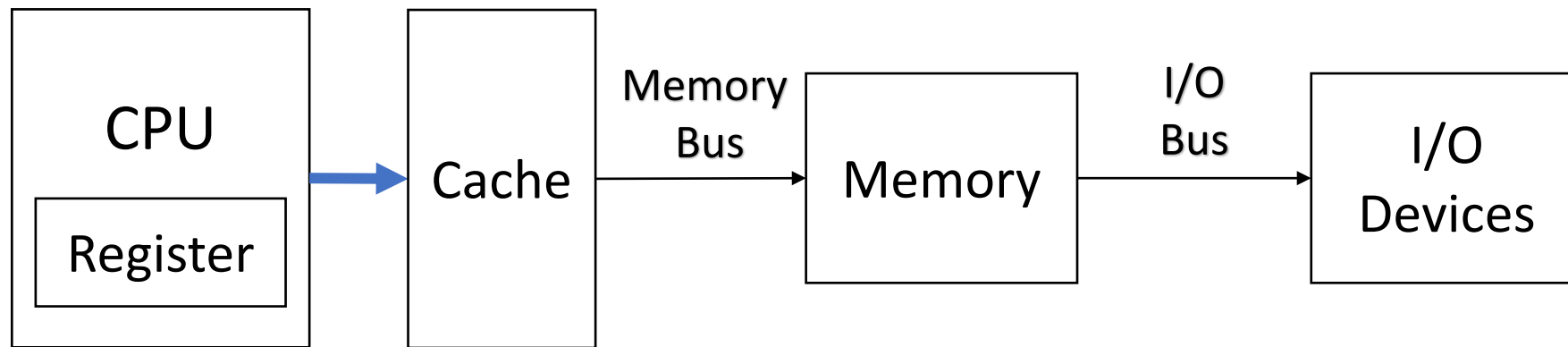
So, what's cache?

- **Cache: a safe place for hiding or storing things.**

**Webster's New World Dictionary of the
American Language
Second College Edition (1976)**



Cache



- The highest or first level of the memory hierarchy encountered once the *addr* leaves the processor
- Employ buffering to reuse commonly occurring items



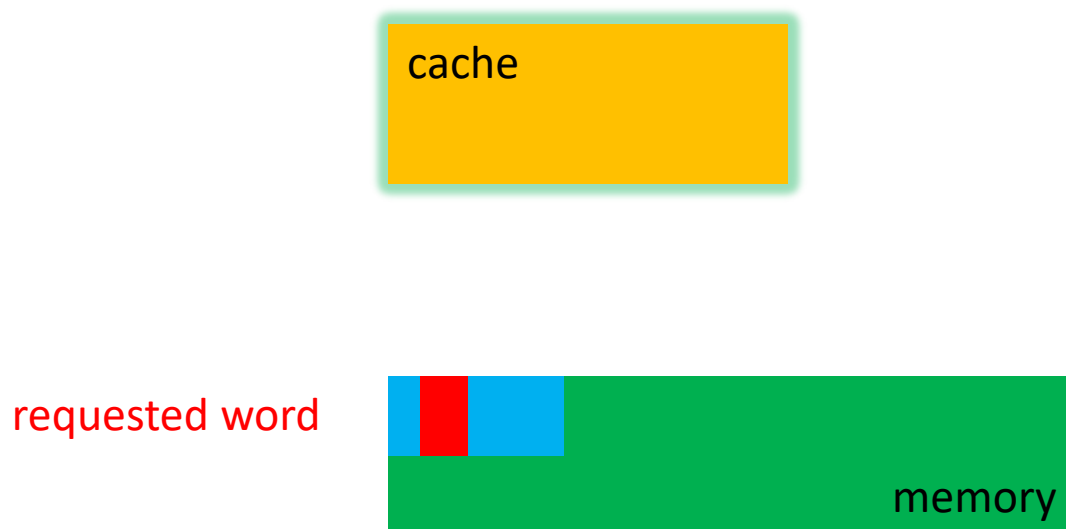
Cache **Hit**/**Miss**

- When the processor **can/cannot** find a requested data item in the cache



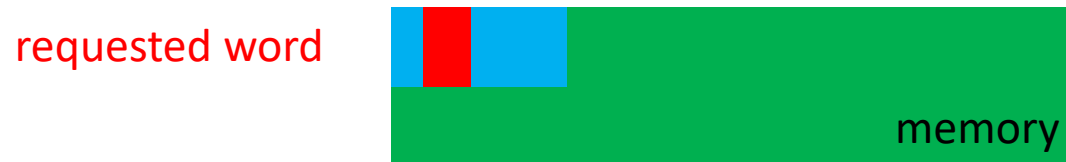
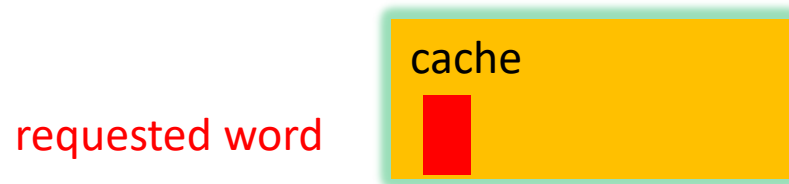
Block/Line Run

- A fixed-size collection of data containing the requested word, retrieved from the main memory and placed into the cache



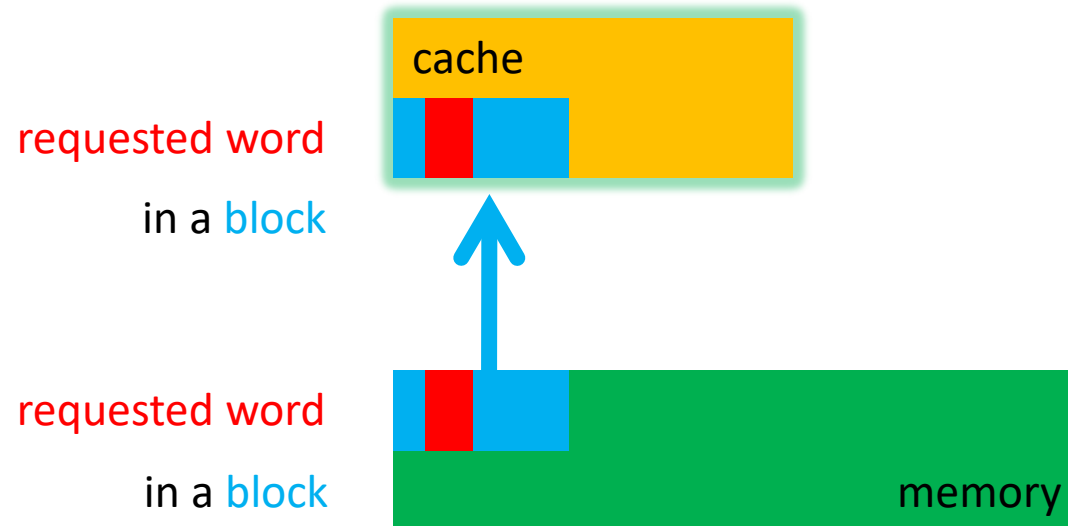
Block/Line Run

- A fixed-size collection of data containing the requested word, retrieved from the main memory and placed into the cache



Block/Line Run

- A fixed-size collection of data containing the requested word, retrieved from the main memory and placed into the cache



Cache Locality

- **Temporal locality**
need the requested word again soon
- **Spatial locality**
likely need other data in the block soon

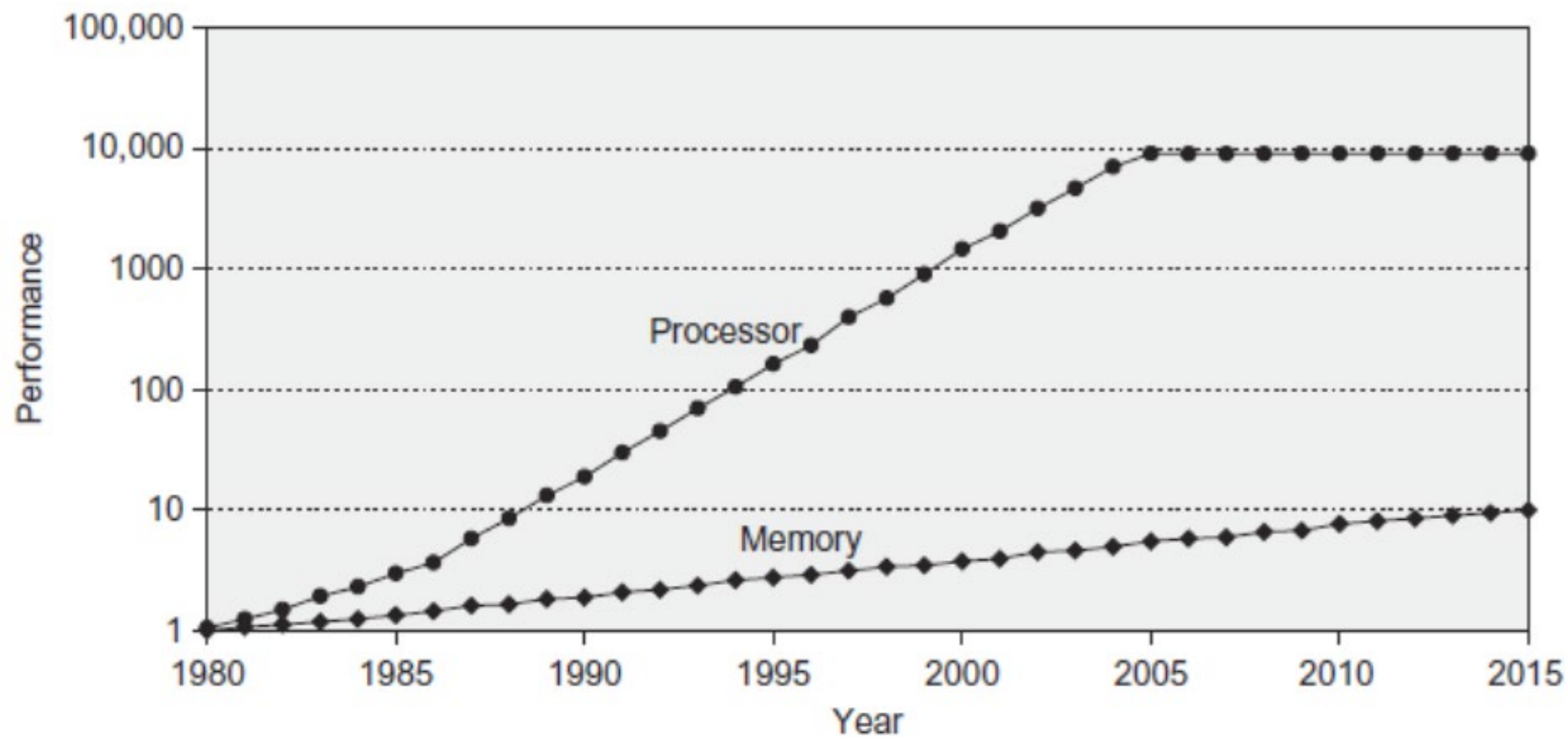


Cache Miss

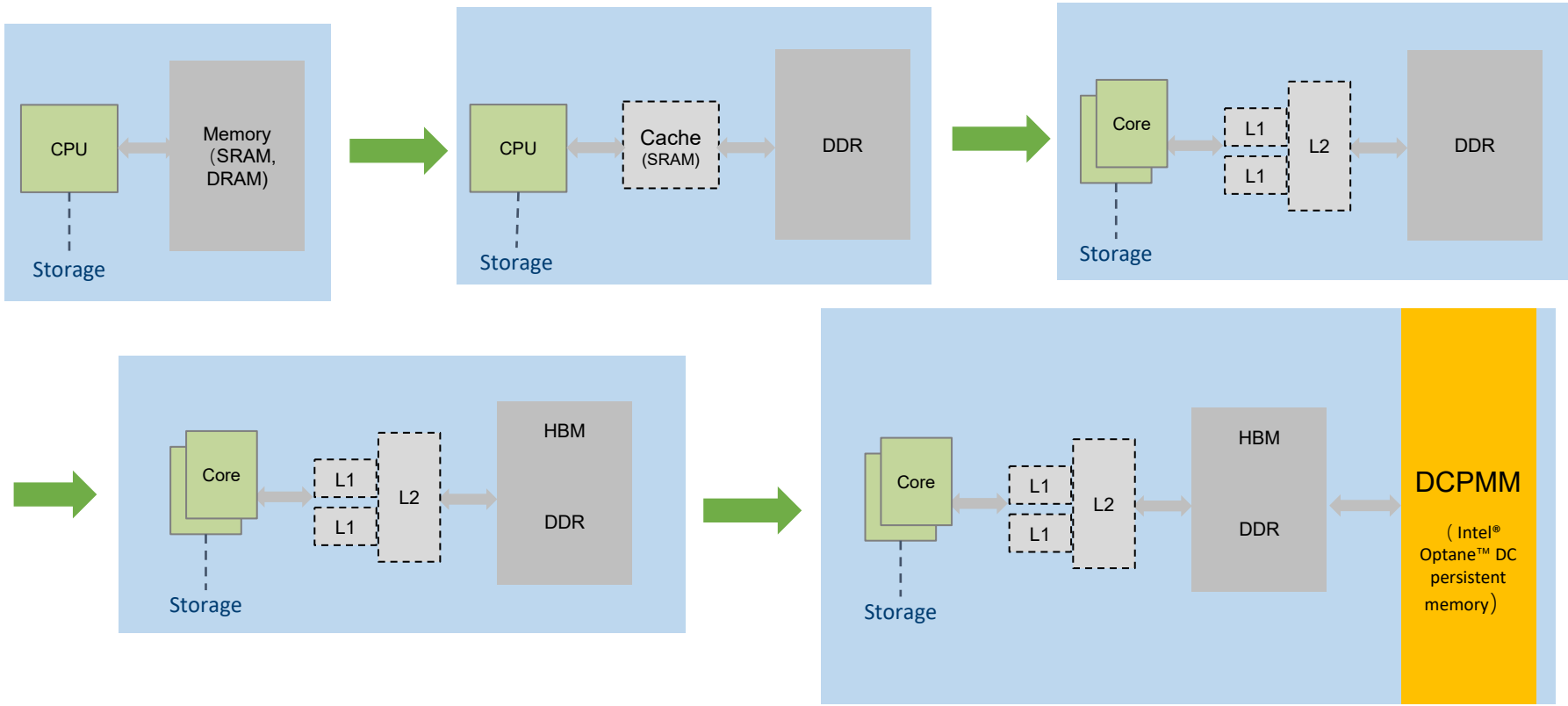
- Time required for cache miss depends on:
Latency: the time to retrieve the first word of the block
Bandwidth: the time to retrieve the rest of this block



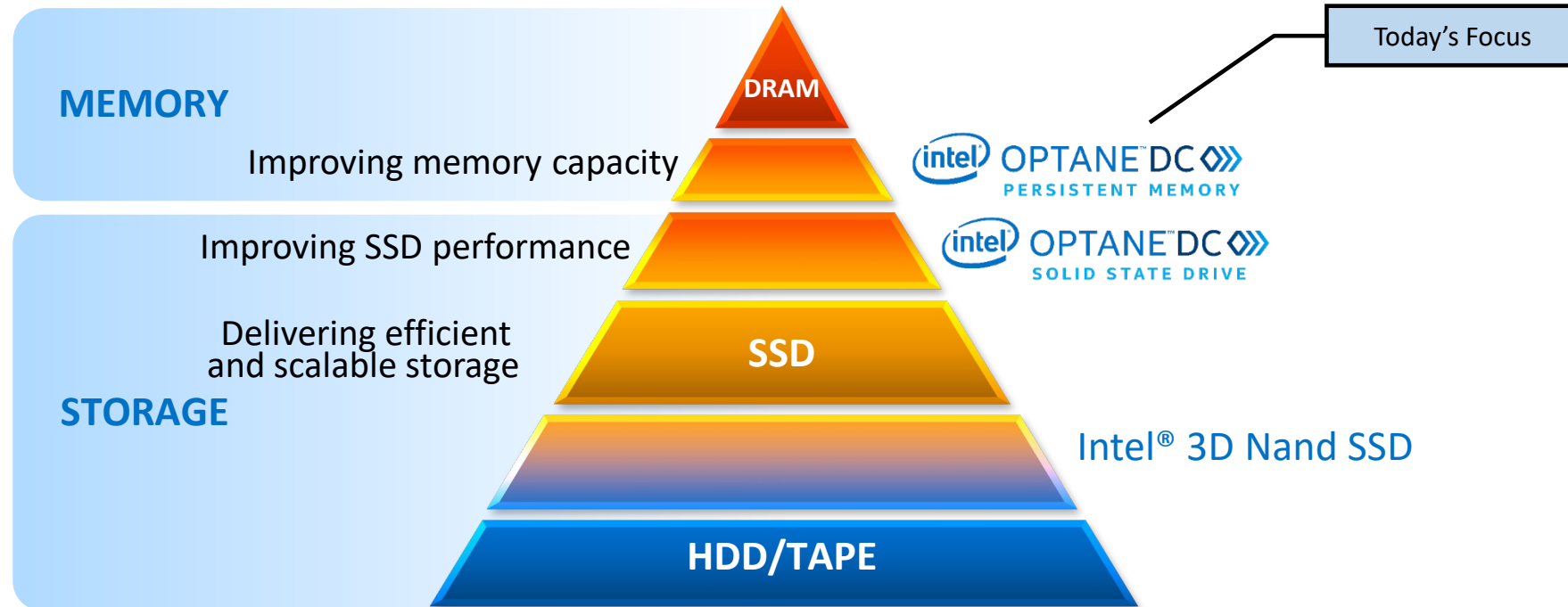
Technology Trend



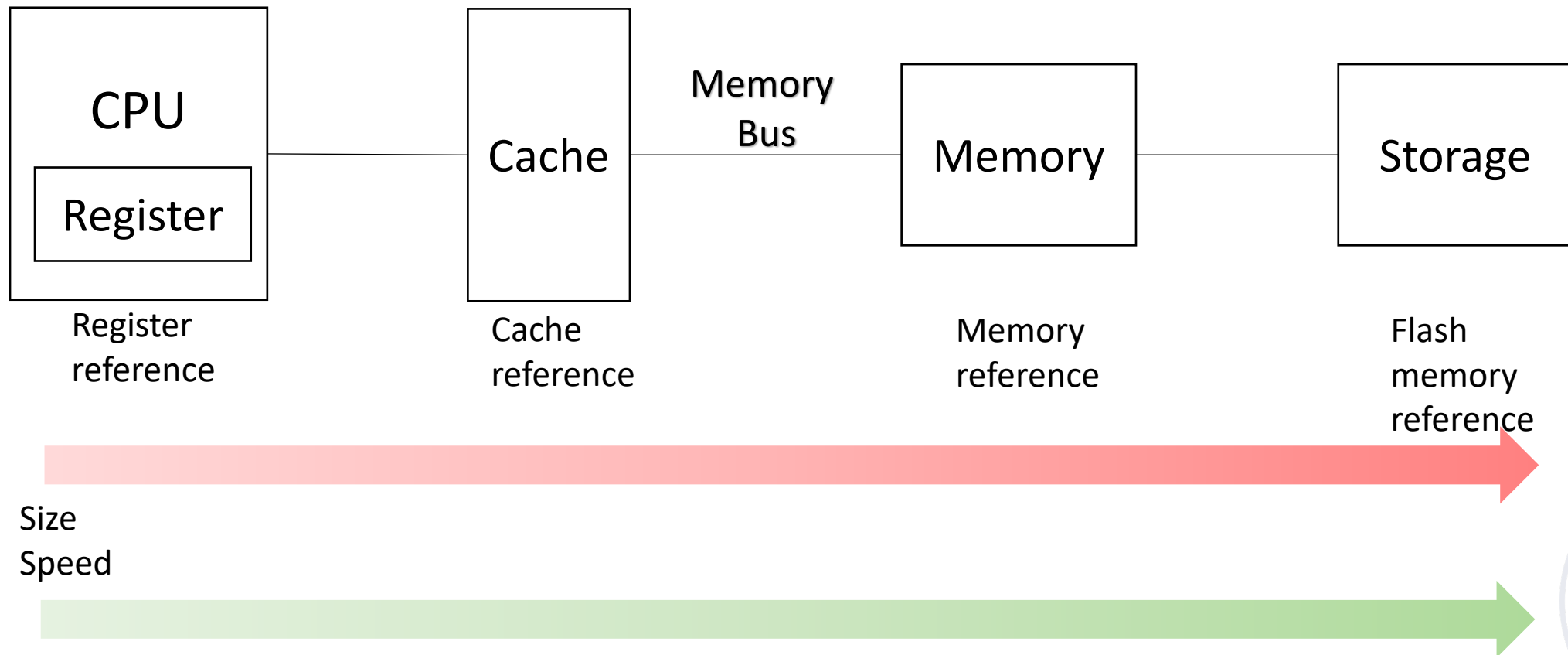
§ 3.2 Technology Trend and Memory Hierarchy



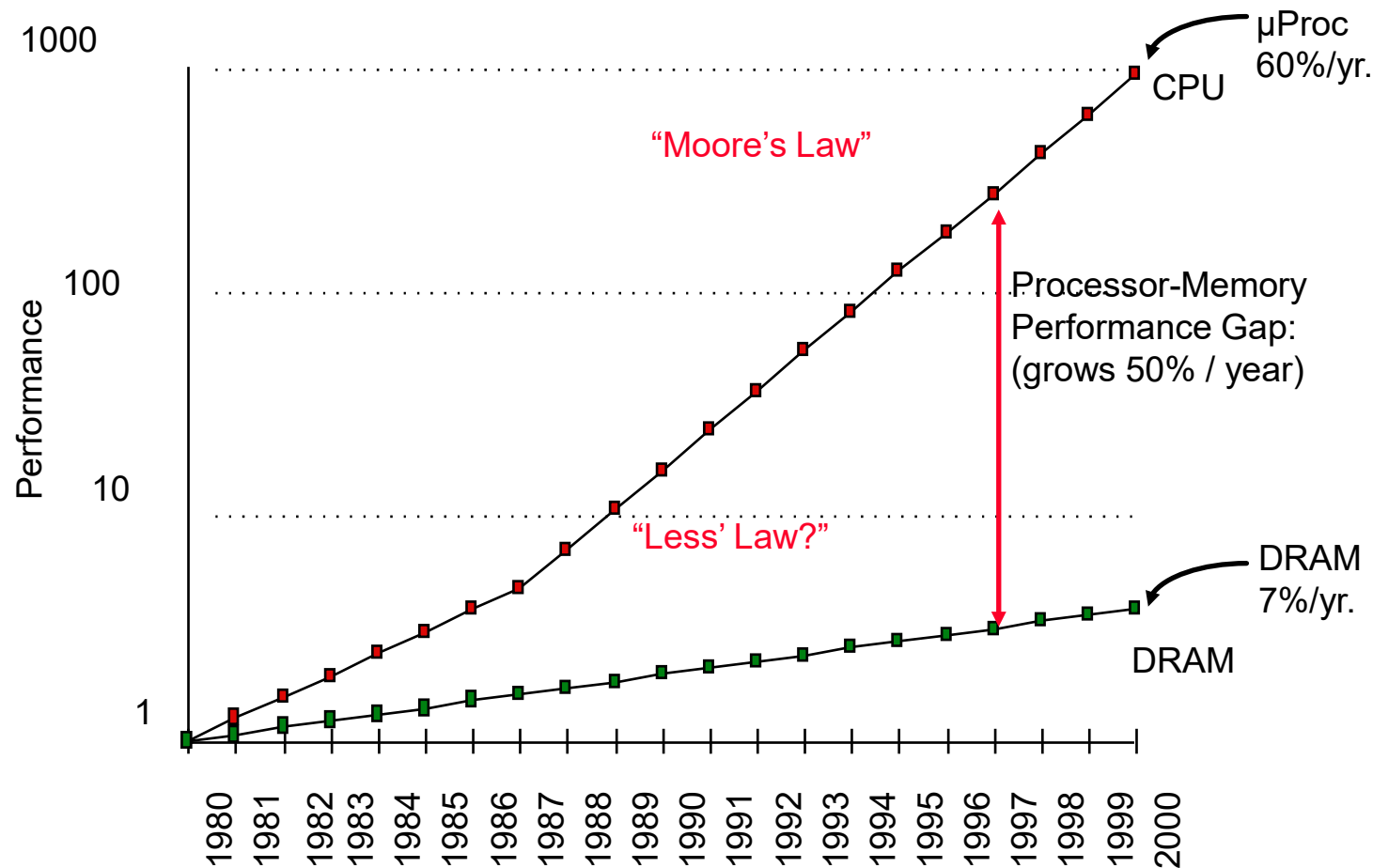
Memory/Storage



Processor-Memory Performance Gap



Who Cares About the Memory Hierarchy?



1980: no cache in μ proc;

1995 2-level cache on chip
(1989 first Intel μ proc with
a cache on chip)



Three classes of computers with different concerns in memory hierarchy

- **Desktop computers:**

- Are primarily running one application for single user
- Are concerned more with average latency from the memory hierarchy.

- **Server computers:**

- May typically have hundreds of users running potentially dozens of applications simultaneously.
- Are concerned about memory bandwidth.



Three classes of computers with different concerns in memory hierarchy.

- **Embedded computers:**
 - Real-time applications.
 - Worst-case performance vs Best case performance
 - Are concerned more about power and battery life.
 - Hardware vs software
 - Running single app & use simple OS
 - The protection role of the memory hierarchy is often diminished.
 - Main memory is very small
 - often no disk storage



Enhance speed of memory

- Component character of hardware:
 - Smaller hardware is faster and more expensive
 - Bigger memories are lower and cheaper

The goal:

- There are speed of smallest memory and capacity of biggest memory
- To provide cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level.



The method enhance speed of memory

By taking advantage of the principle of locality:

- **most programs do not access all code or data uniformly**
- **Temporal Locality** (Locality in Time):
 - If an item is referenced, the **same item** will tend to be referenced again **soon**
 - Keep most recently accessed data items closer to the processor
- **Spatial Locality** (Locality in Space):
 - If an item is referenced, **nearby items** will tend to be referenced **soon**
 - Move recently accessed groups of contiguous words(block) closer to processor.



The method enhance speed of memory

The method

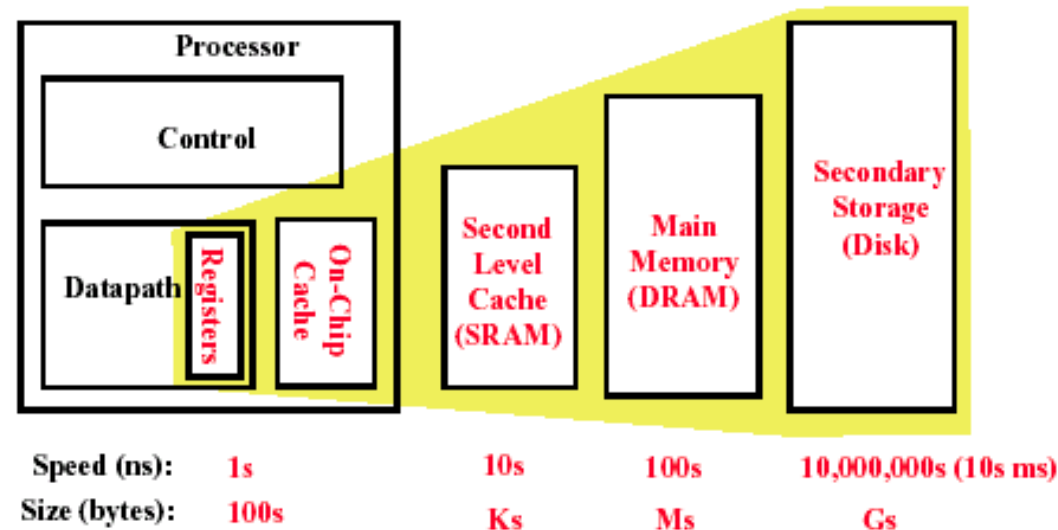
- Hierarchies bases on memories of different speeds and size
- The more closely CPU the level is, the faster the one is.
- The more closely CPU the level is, the smaller the one is.
- The more closely CPU the level is, the more expensive one is.



Memory Hierarchy of a Modern Computer System

By taking advantage of the principle of locality:

- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.



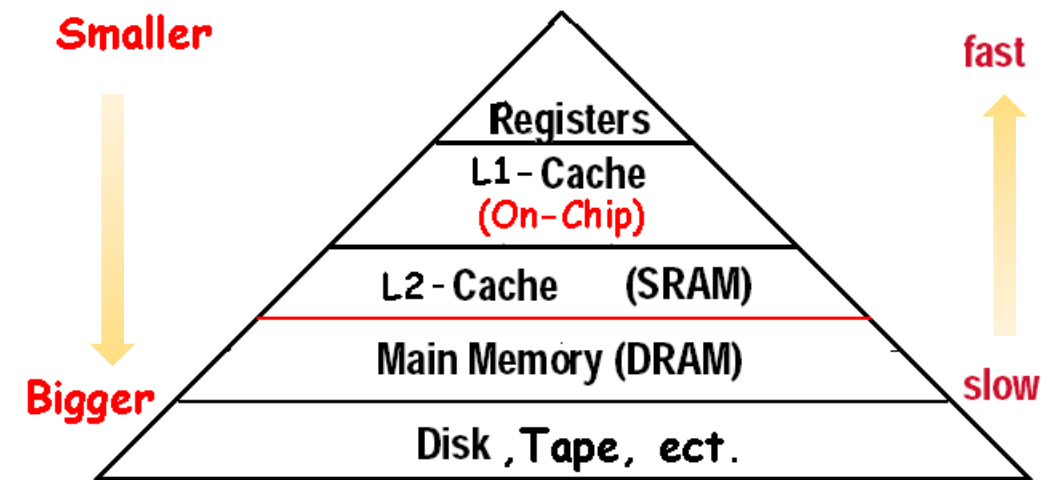
36 terms of Cache

Cache	full associative	write allocate
Virtual memory	dirty bit	unified cache
Memory stall cycles	block	block offset
misses per instruction	direct mapped	write back
Valid bit	data cache	locality
Block address	hit time	address trace
Write through	cache miss	set
Instruction cache	page fault	miss rate
random replacement	index field	cache hit
Average memory access time	page	tag field
n-way set associative	no-write allocate	miss penalty
Least-recently used	write buffer	write stall

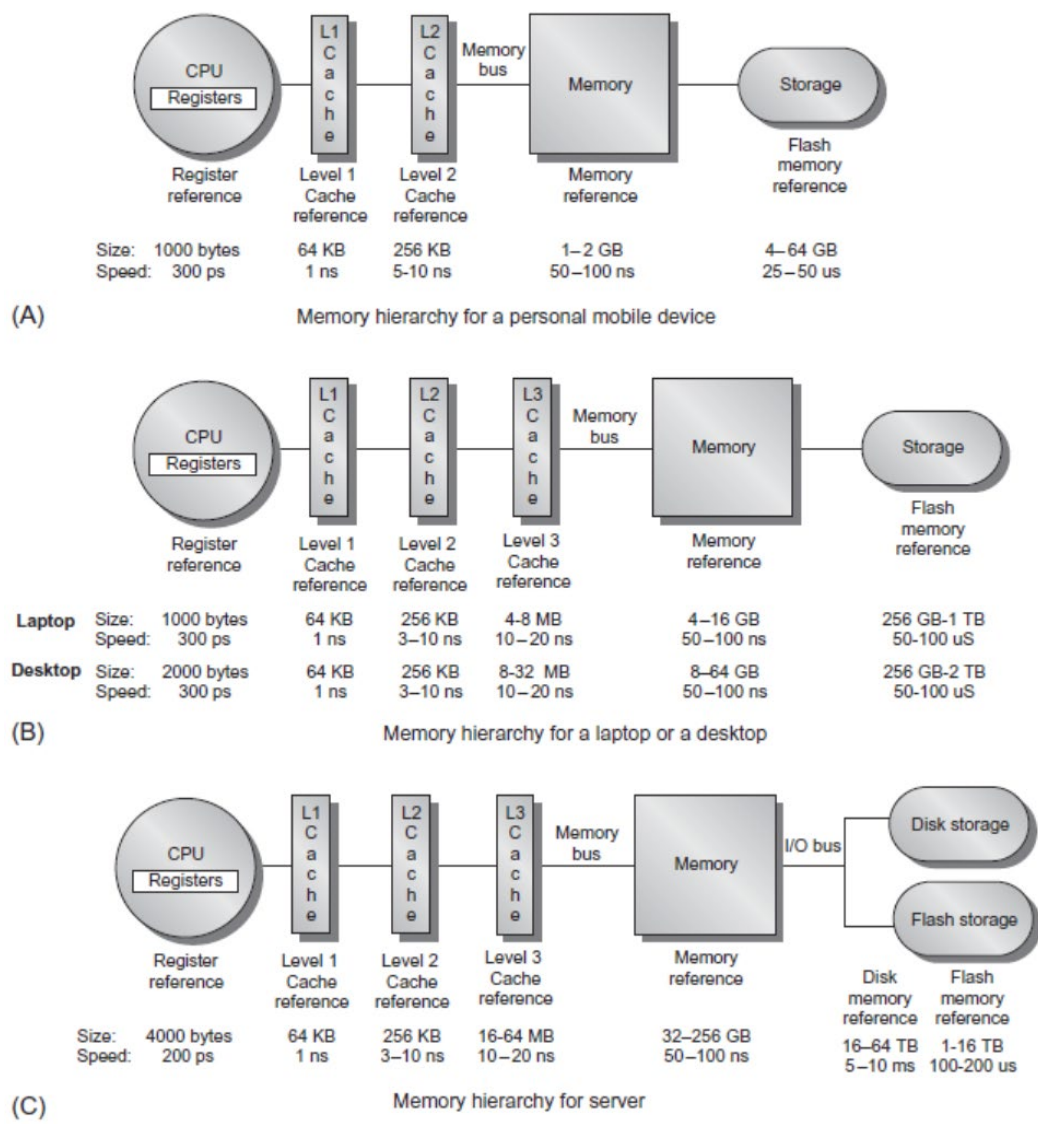


What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- In computer architecture, almost everything is a cache!
 - Registers “a cache” on variables – software managed
 - First-level cache a cache on second-level cache
 - Second-level cache a cache on memory
 - Memory a cache on disk (virtual memory)
 - TLB a cache on page table
 - Branch-prediction a cache on prediction information?



§ 3.2 Technology Trend and Memory Hierarchy



Four Questions for Cache Designers

Caching is a general concept used in processors, operating systems, file systems, and applications.

There are **Four Questions** for Cache/Memory Hierarchy Designers

- **Q1:** Where can a block be placed in **the upper level/main memory**?
(Block placement)
 - Fully Associative, Set Associative, Direct Mapped
- **Q2:** How is a block found if it is in **the upper level/main memory**?
(Block identification)
 - Tag/Block
- **Q3:** Which block should be replaced on a **Cache/main memory** miss?
(Block replacement)
 - Random, LRU, FIFO
- **Q4:** What happens on a write?
(Write strategy)
 - Write Back or Write Through (with Write Buffer)

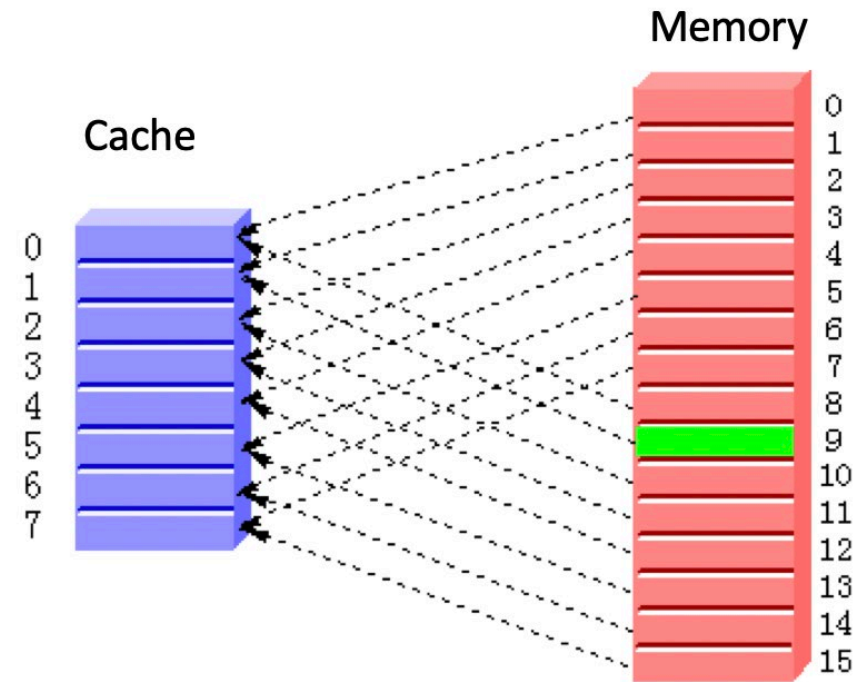
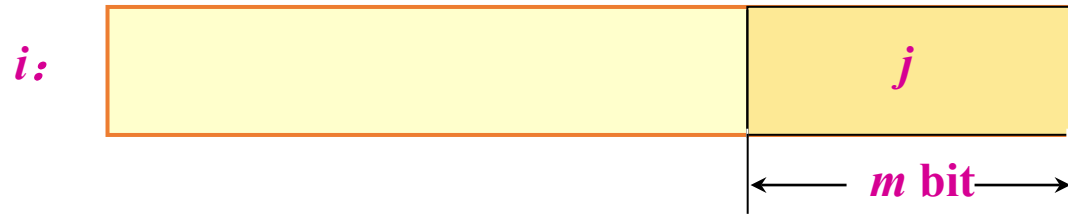


Q1: Block Placement

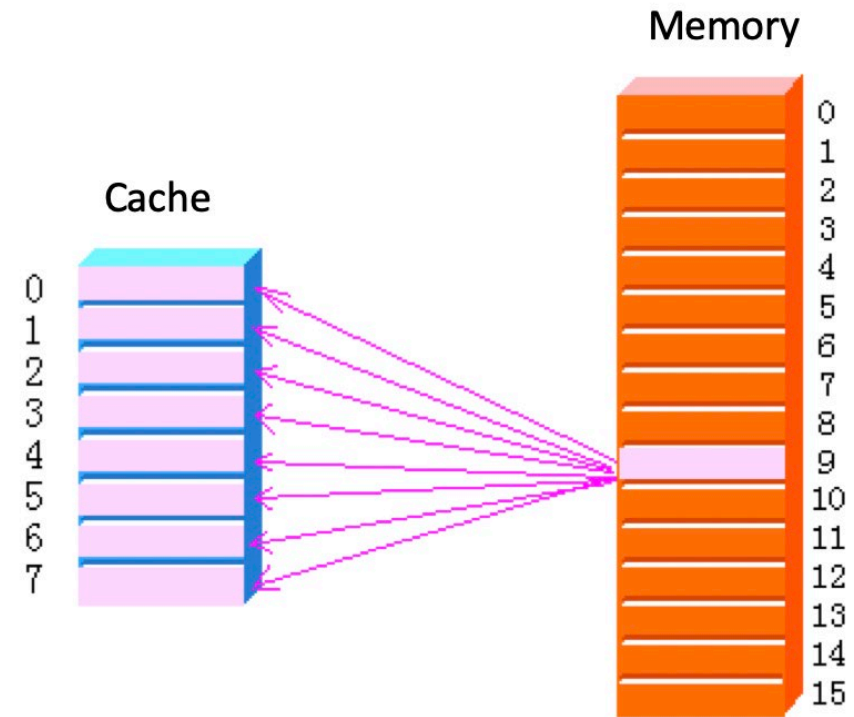
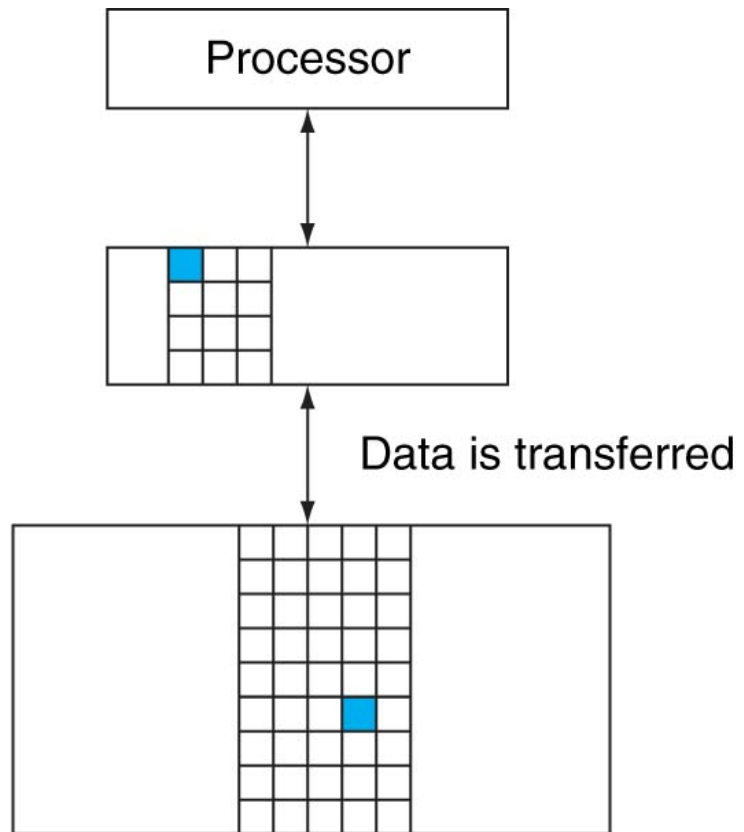
- Direct mapped
 - Block can only go in one place in the cache.
- Fully associative
 - *Note that direct mapped is the same as 1-way set associative, and fully associative is m-way set-associative (for a cache with m blocks).*
- Set associative
 - Block can go in one of a set of places in the cache.
 - A set is a group of blocks in the cache.
Block address MOD Number of sets in the cache
 - If sets have n blocks, the cache is said to be n-way set associative.



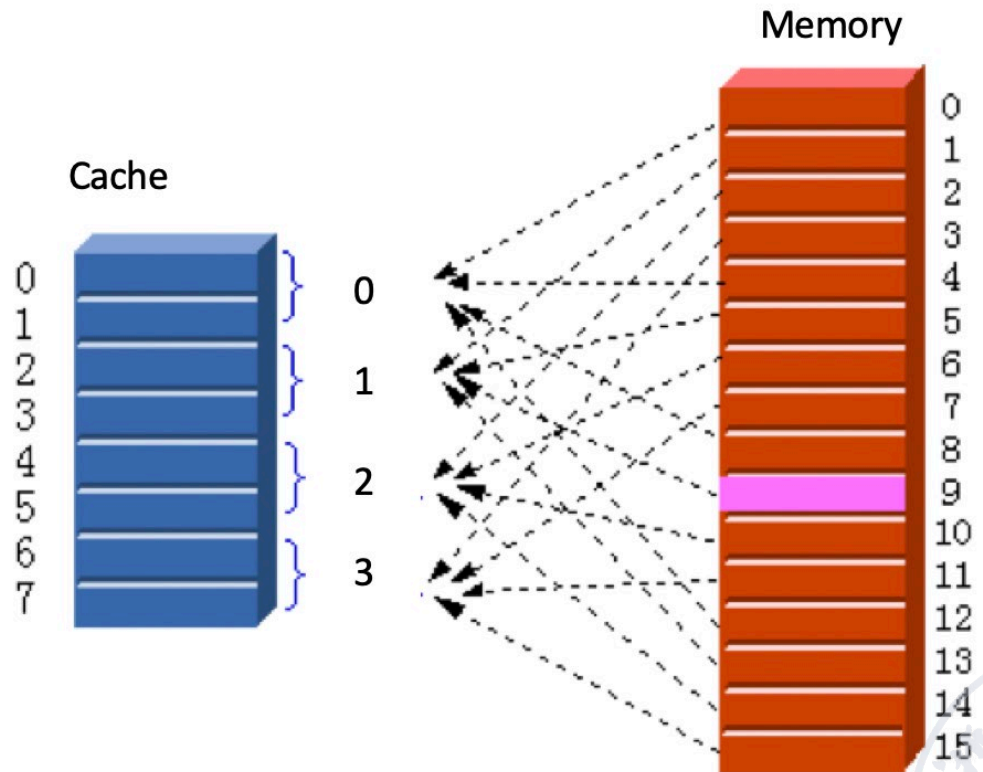
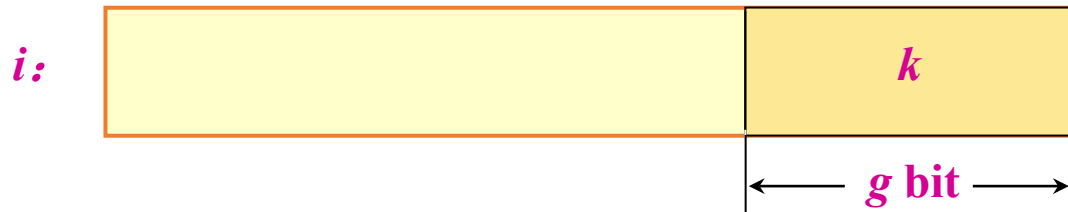
Direct mapped



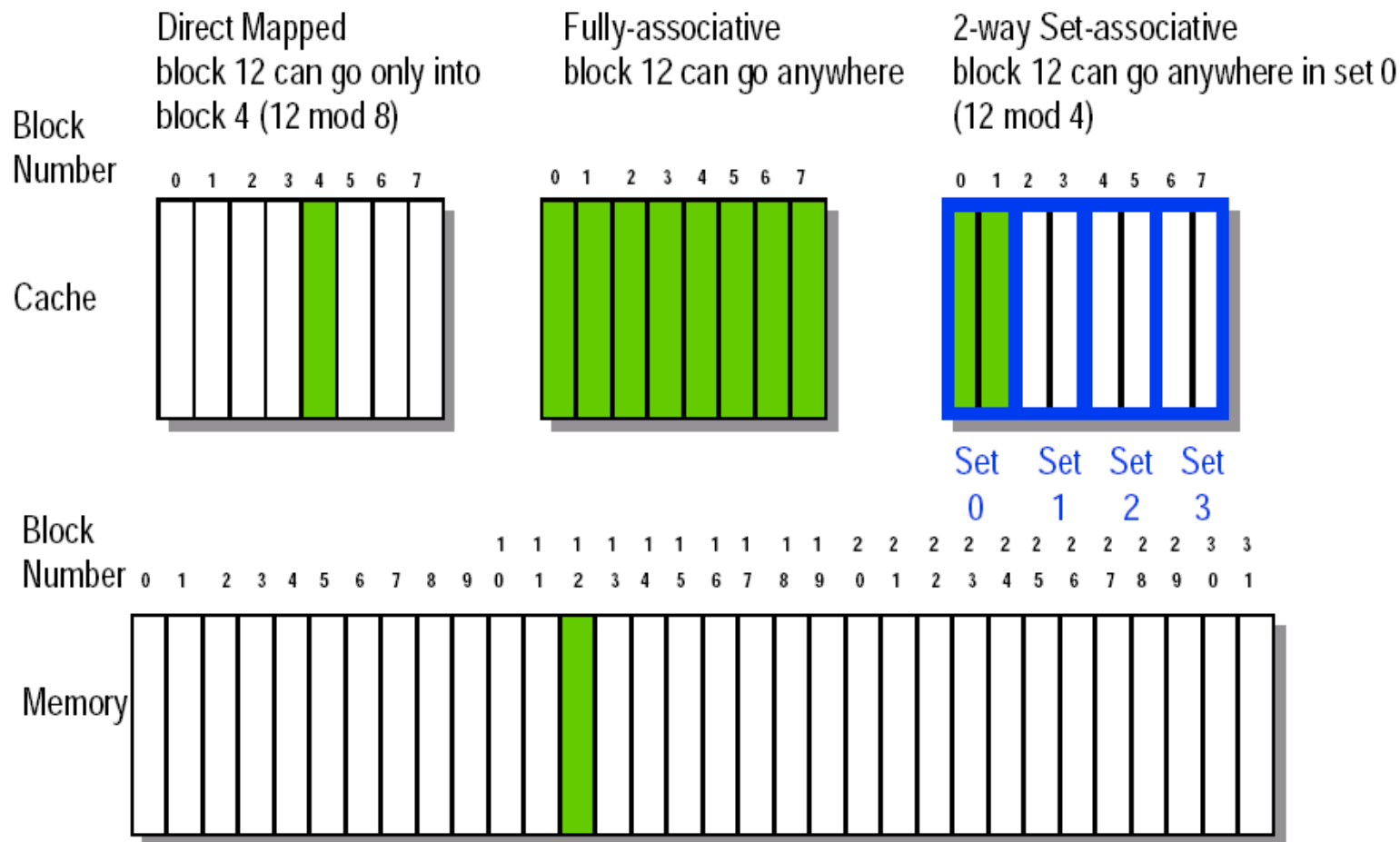
Fully-associative



2-way Set-associative



8-32 Block Placement



N-way Set-associative

- The higher the degree of association, the higher the utilization of cache space, the lower the probability of block collision and the lower the failure rate.

	n	G
Full-associative	M	1
Direct mapped	1	M
Set-associative	$1 < n < M$	$1 < G < M$

- Most Cache: $n \leq 4$
- Question: Is the greater the number n , the better?



Q2: Block Identification

- Every block has an **address tag** that stores the main memory address of the data stored in the block.
- When checking the cache, the processor will **compare** the requested **memory address to the cache tag** -- if the two are equal, then there is a cache hit and the data is present in the cache
- Often, each cache block also has a **valid bit** that tells if the contents of the cache block are valid



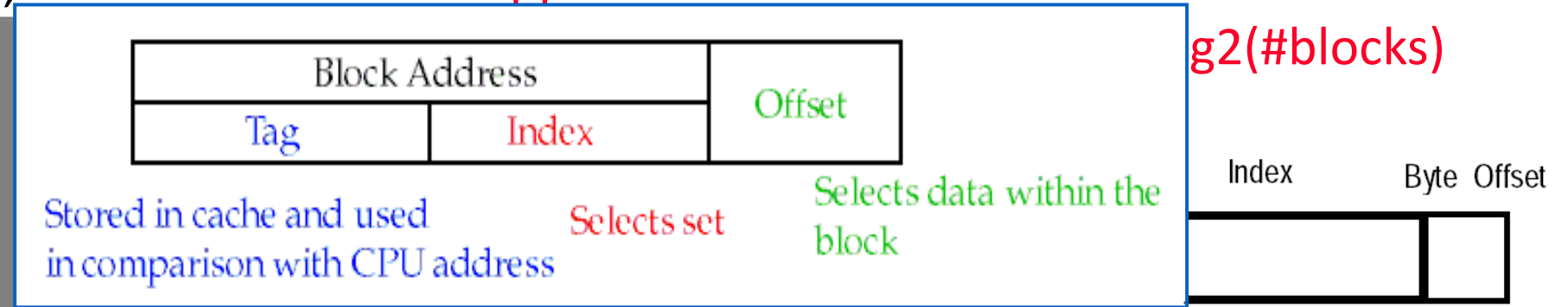
The Format of the Physical Address

- The **Index** field selects
 - The **set**, in case of a **set-associative cache**
 - The **block**, in case of a **direct-mapped cache**
 - Has as many bits as $\log_2(\#blocks)$ for **direct-mapped**

- The **Byte Offset**

- The byte offset
 - Has as many bits as $\log_2(size\ of\ block)$

- The **Tag** is used to find the matching block within a set or in the cache
 - Has as many bits as $Address_size - Index_size - Byte_Offset_Size$

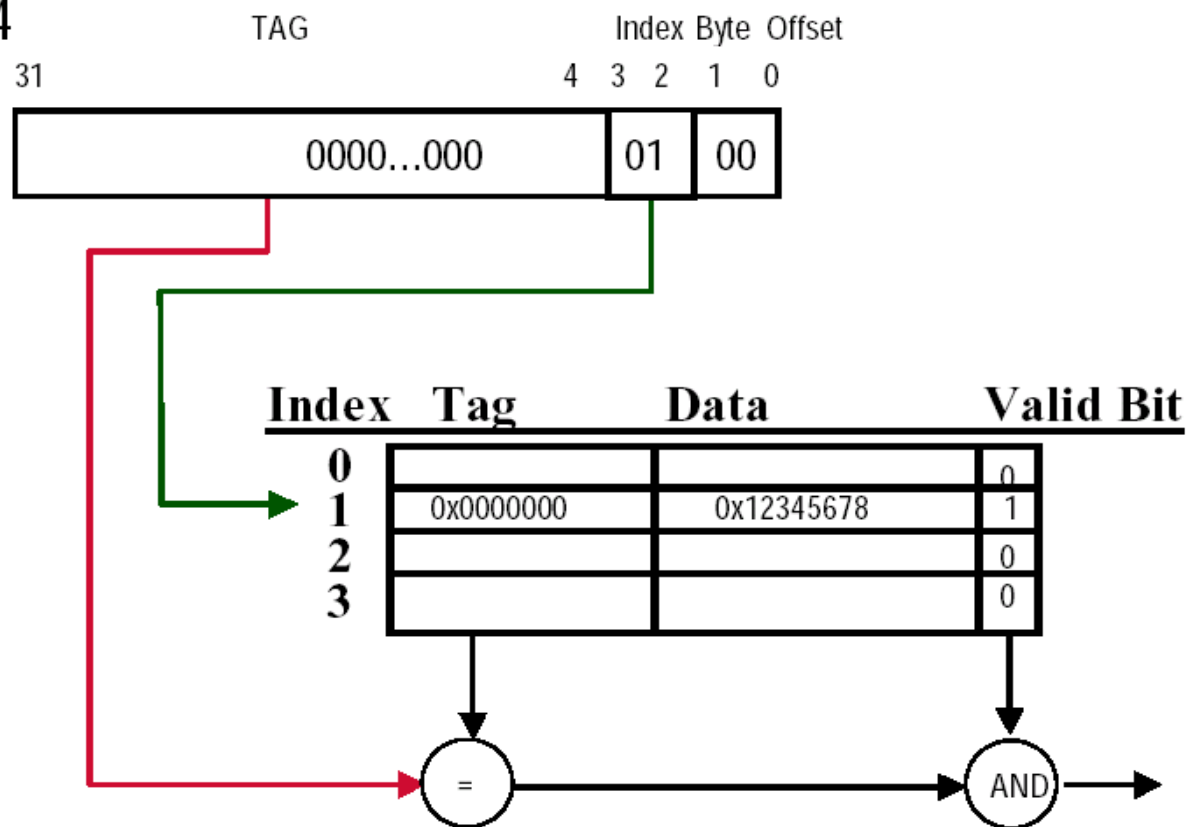


Direct-mapped Cache Example (1-word Blocks)

LOAD R1, 0x04

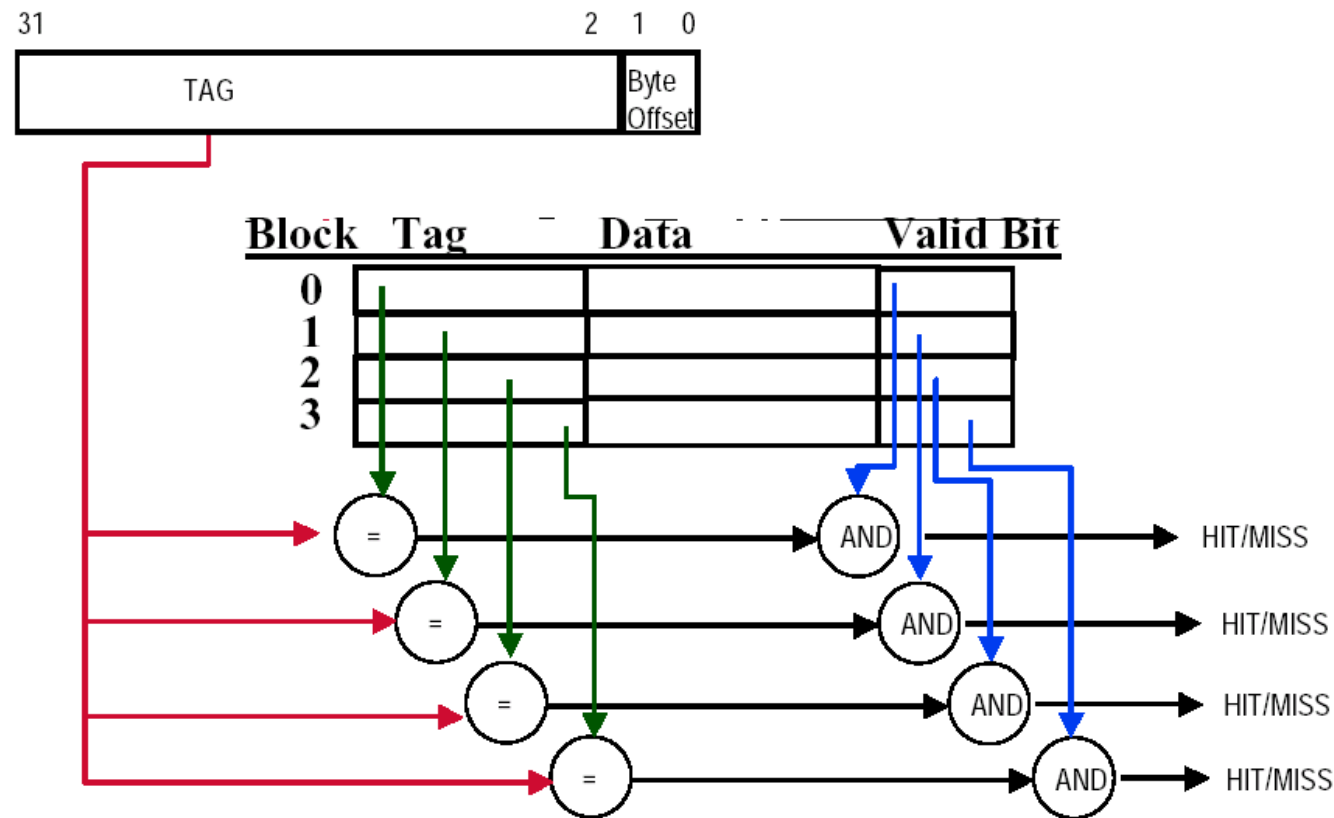
MEMORY

Address	Data
0x00	0x00000000
0x04	0x12345678
0x08	0x87654321
0x0C	0x11111111
0x10	0x22222222
0x14	0x33333333
0x18	0x44444444
0x1C	0x55555555
0x20	0x10101010



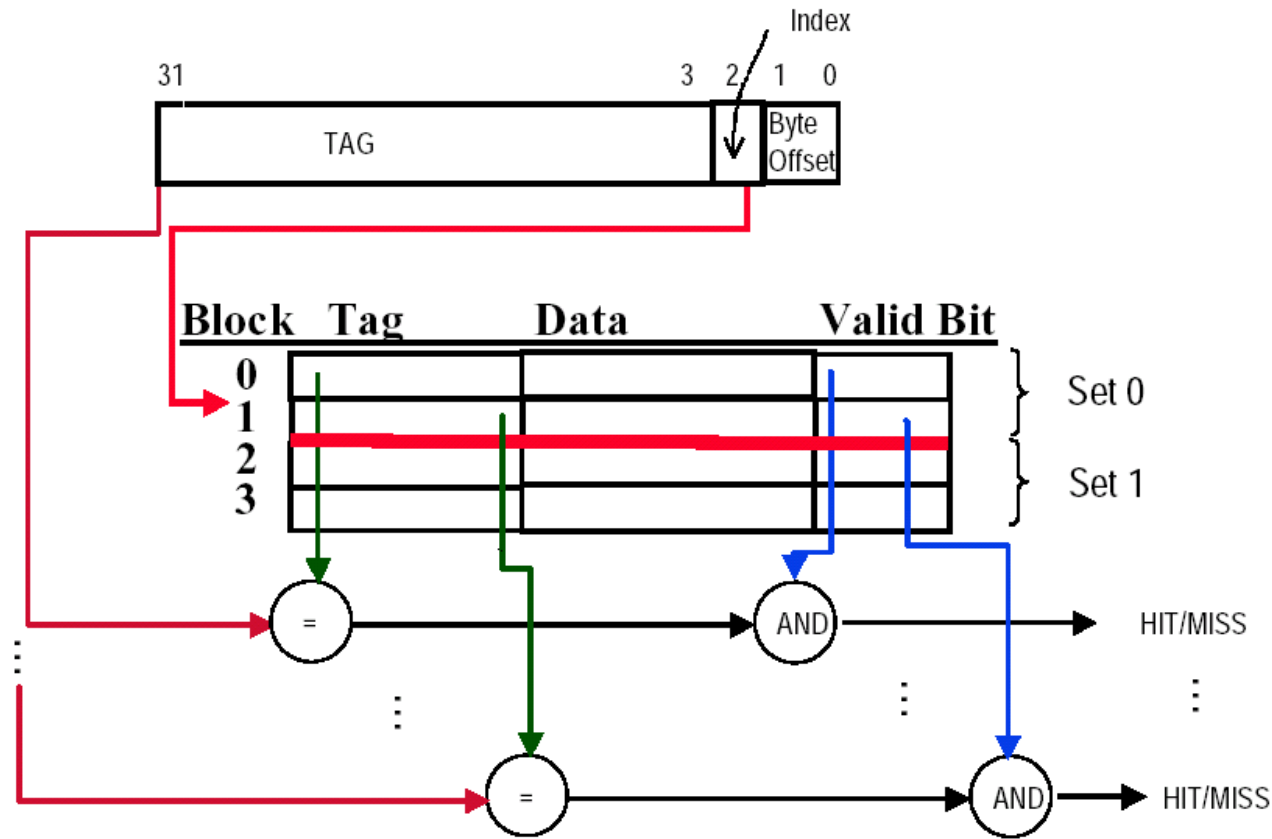
Fully-Associative Cache example (1-word Blocks)

- Assume cache has 4 blocks



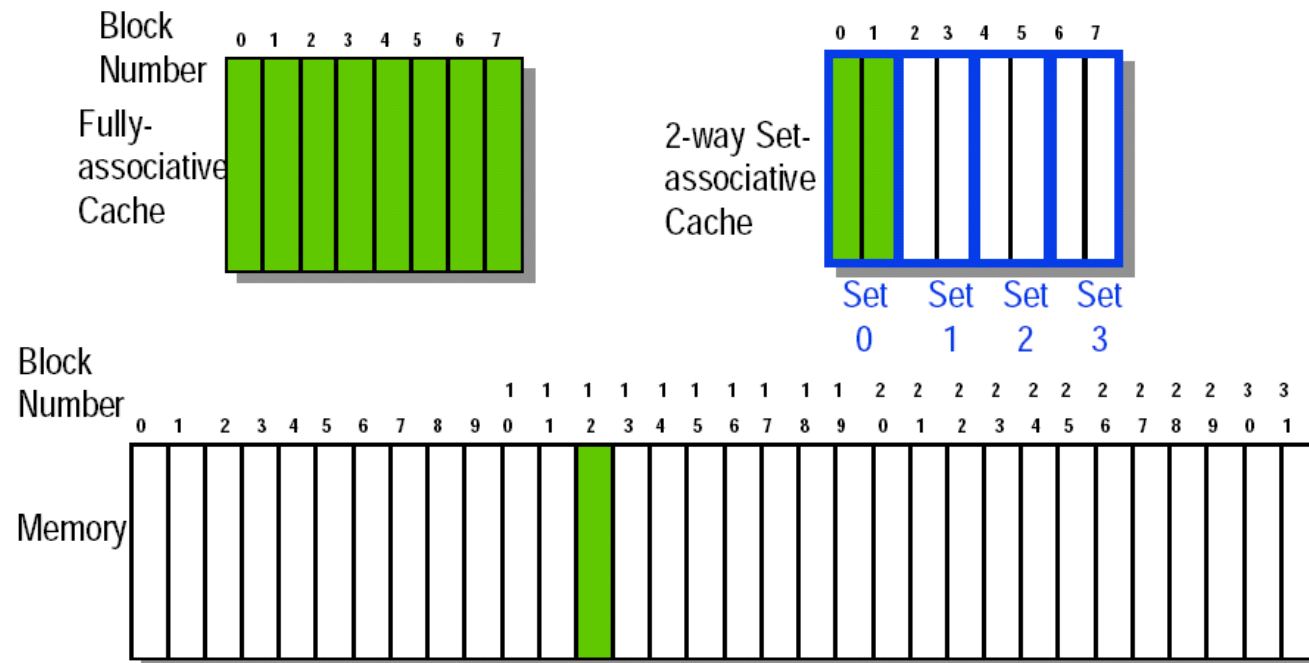
2-Way Set-Associative Cache

- Assume cache has 4 blocks and each block is 1 word
- 2 blocks per set, hence 2 sets per cache



Q3: Block Replacement

- In a direct-mapped cache, there is only one block that can be replaced
- In set-associative and fully-associative caches, there are N blocks (where N is the degree of associativity)



Strategy of Block Replacement

- Several different replacement policies can be used
 - **Random replacement** - *randomly pick any block*
 - Easy to implement in hardware, just requires a random number generator
 - Spreads allocation uniformly across cache
 - May evict a block that is about to be accessed
 - **Least-Recently Used (LRU)** - *pick the block in the set which was least recently accessed*
 - Assumed more recently accessed blocks more likely to be referenced again
 - This requires extra bits in the cache to keep track of accesses.
 - **First In, First Out (FIFO)** - *Choose a block from the set which was first came into the cache*



Strategy of Block Replacement

Suppose:

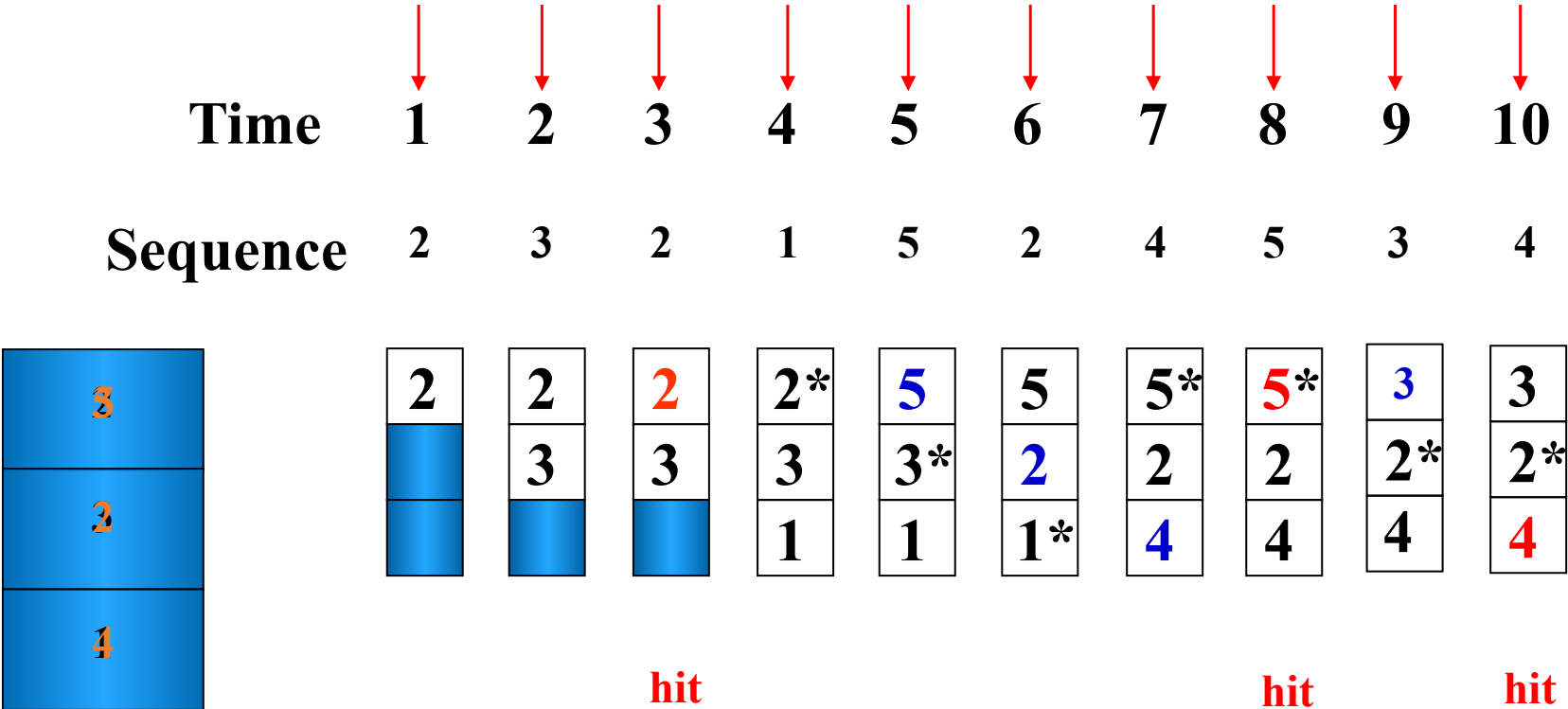
- Cache block size is 3, and access sequence is shown as follows.

2, 3, 2, 1, 5, 2, 4, 5, 3, 4

- FIFO, LRU and OPT are used to simulate the use and replacement of cache block.



FIFO

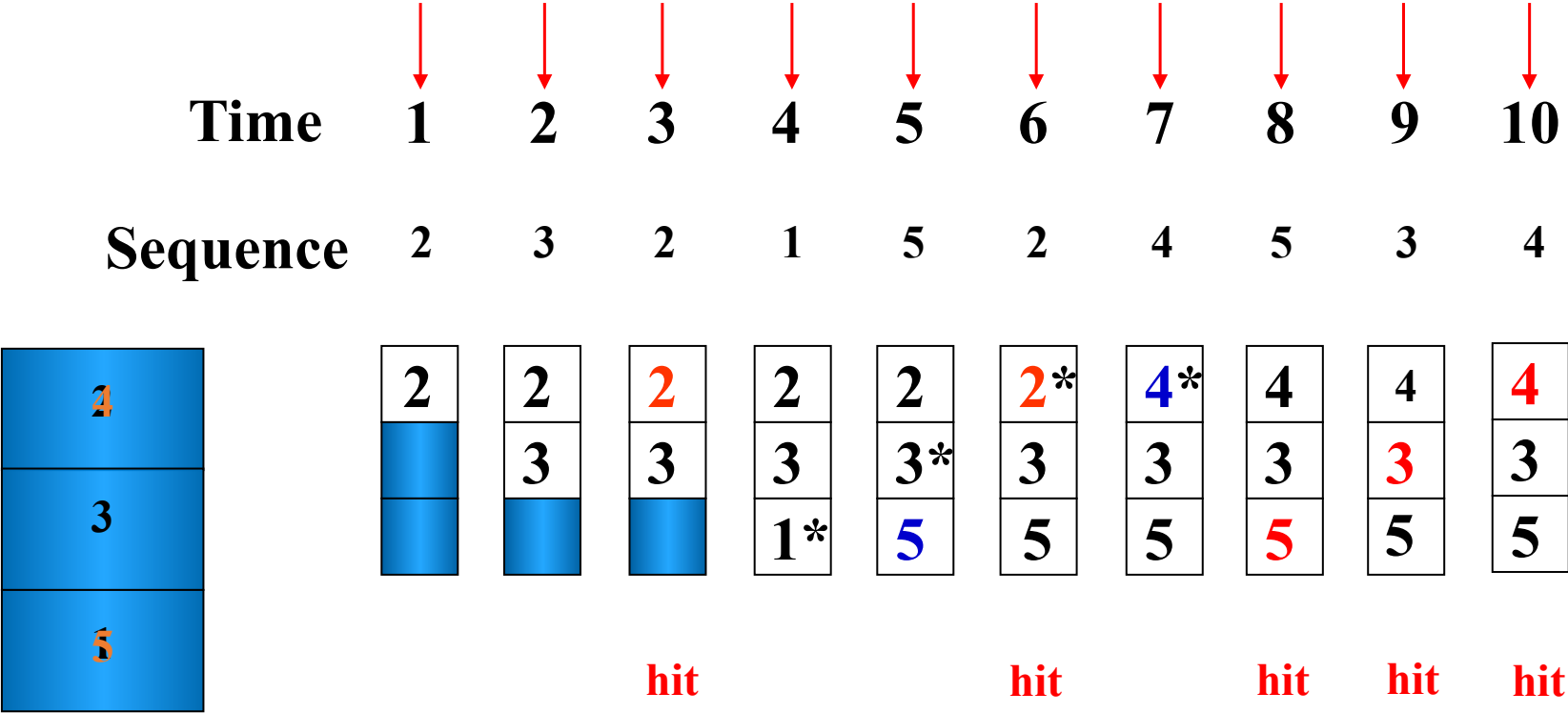


LRU

		↓	↓	↓	↓	↓	↓	↓	↓	↓	
Time		1	2	3	4	5	6	7	8	9	10
Sequence		2	3	2	1	5	2	4	5	3	4
	3	2	2	2	2	2*	2	2	2*	3	3
	3		3	3*	5	5	5*	5	5	5	5*
	4				1	1	1*	4	4	4*	4
				hit		hit		hit		hit	



OPT



The hit rate is related to the replacement algorithm.



Time Sequence	1	2	3	4	5	6	7	8	thrashing
	1	2	3	4	1	2	3	4	
FIFO	1	1	1*	4	4	4*	3	3	No hit
n=3		2	2	2*	1	1	1*	4	
			3	3	3*	2	2	2*	
LRU	1	1	1*	4	4	4*	3	3	Hit 4 times
n=4		2	2	2*	2	2	2*	2	
			3	3	3*	3	3	3*	
OPT	1	1	1	4	4*	4	4	4	Hit 3 times
		2	2	2	2	2*	3*	3	
			3*	4*	4	4	4	4*	

➤ Hit rate is related to access sequence.



Stack replacement algorithm

- $B_t(n)$ represents the set of access sequences contained in a cache block of size n at time t .
- $B_t(n)$ is the subset of $B_t(n + 1)$.



LRU replacement algorithm is stack replacement algorithm

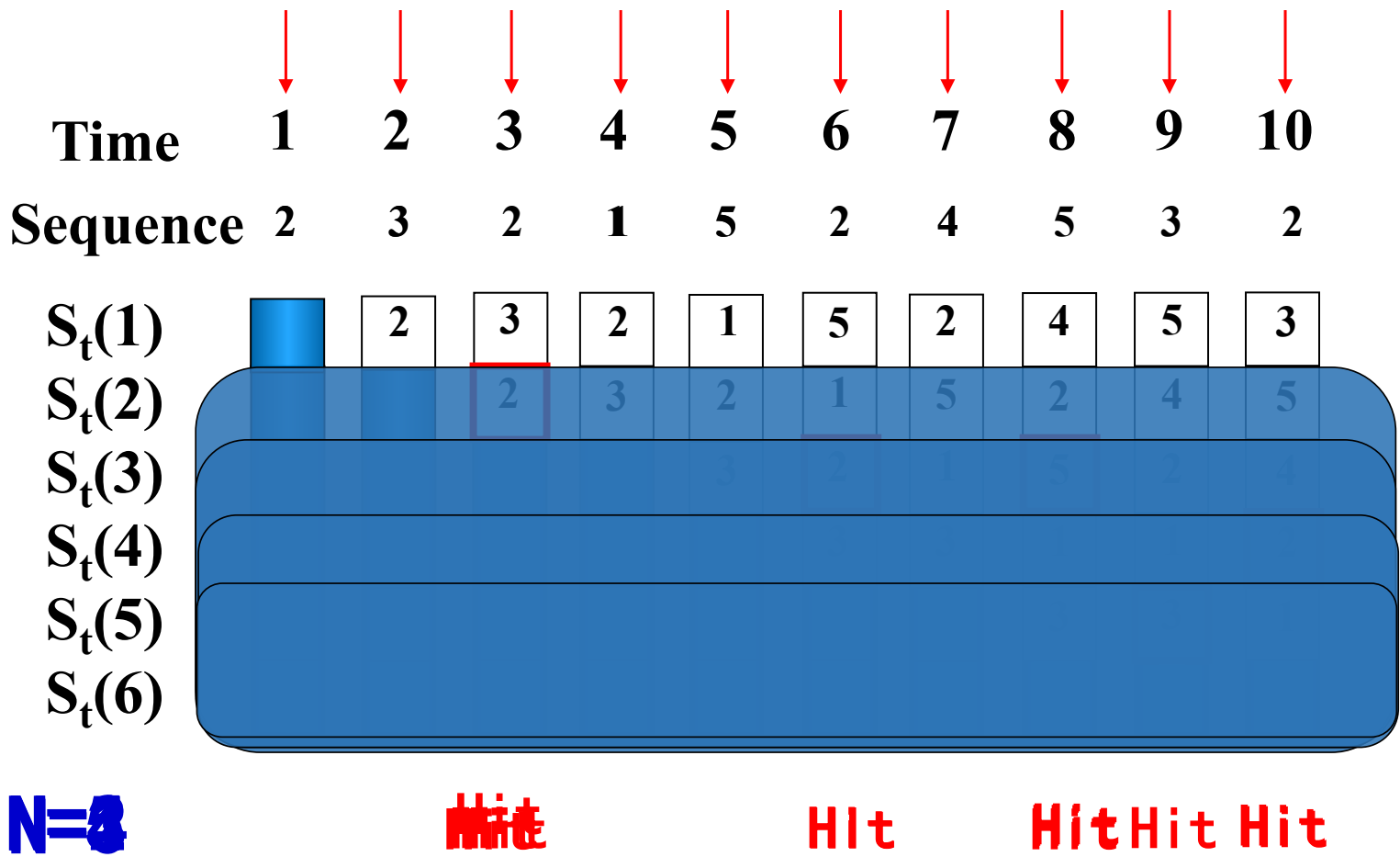
Time	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	1	2	3	4	1	2	5	1	2	3	4	5
LRU n=3 Bt(n)	<div><div>1</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div></div>	<div><div>1*</div><div>2</div><div>3</div></div>	<div><div>4</div><div>2*</div><div>3</div></div>	<div><div>4</div><div>1</div><div>3*</div></div>	<div><div>4*</div><div>1</div><div>2</div></div>	<div><div>5</div><div>1*</div><div>2</div></div>	<div><div>5</div><div>1</div><div>2*</div></div>	<div><div>5*</div><div>1</div><div>2</div></div>	<div><div>3</div><div>1*</div><div>2</div></div>	<div><div>3</div><div>4</div><div>2*</div></div>	<div><div>3*</div><div>4</div><div>5</div></div>
LRU n+1=4 Bt(n+1)	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>1*</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2*</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3*</div><div>4</div></div>	<div><div>1</div><div>2</div><div>5</div><div>4*</div></div>	<div><div>1</div><div>2</div><div>5</div><div>4*</div></div>	<div><div>1</div><div>2</div><div>5</div><div>4*</div></div>	<div><div>1</div><div>2</div><div>5*</div><div>3</div></div>	<div><div>1*</div><div>2</div><div>4</div><div>3</div></div>	<div><div>5</div><div>2*</div><div>4</div><div>3</div></div>

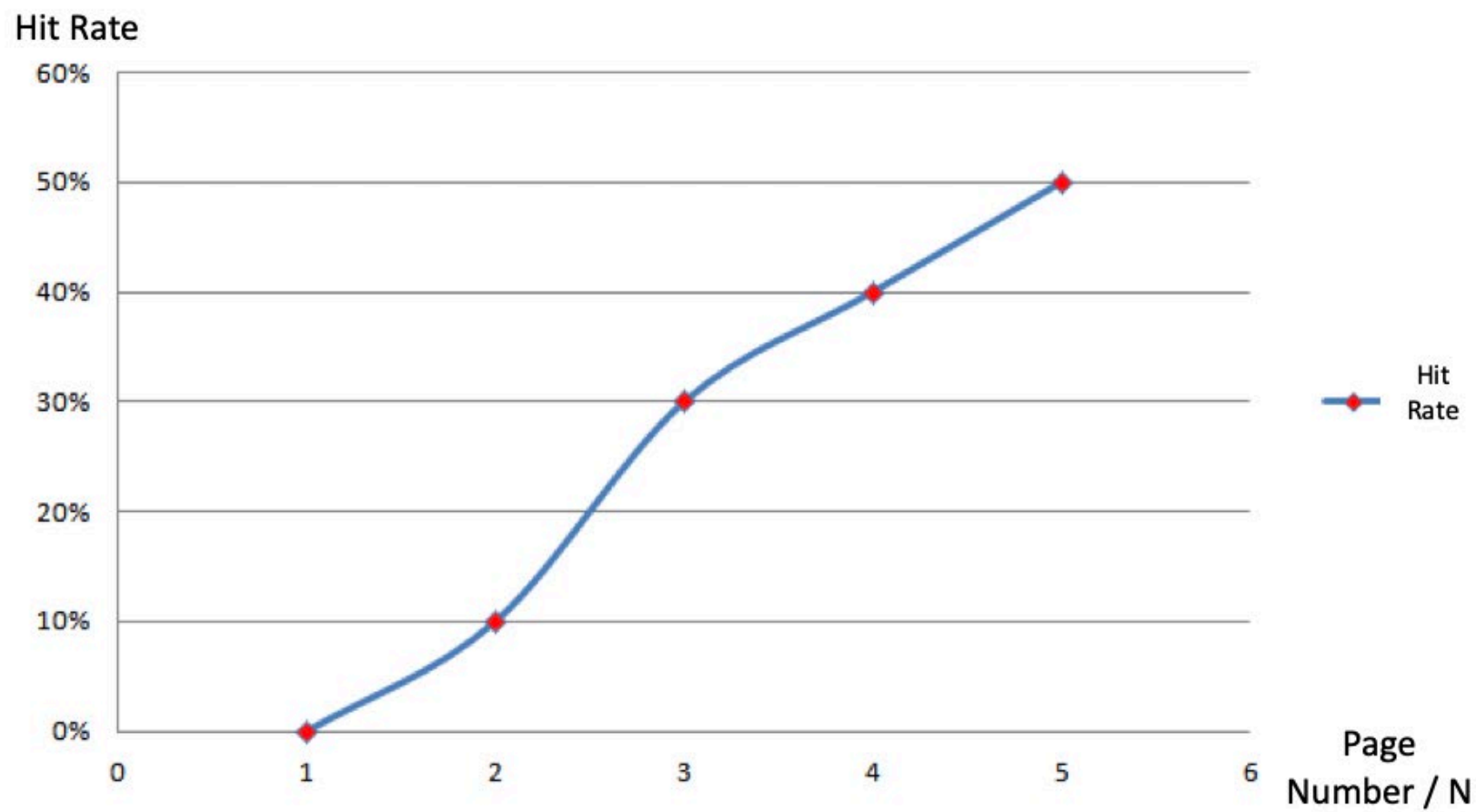


Time	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	1	2	3	4	1	2	5	1	2	3	4	5
FIFO n=3	<div><div>1</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div></div>	<div><div>1*</div><div>2</div><div>3</div></div>	<div><div>4</div><div>2*</div><div>3</div></div>	<div><div>4</div><div>1</div><div>3*</div></div>	<div><div>4*</div><div>1</div><div>2</div></div>	<div><div>5</div><div>1*</div><div>2</div></div>	<div><div>5</div><div>1*</div><div>2</div></div>	<div><div>5</div><div>1*</div><div>2</div></div>	<div><div>5</div><div>3</div><div>2*</div></div>	<div><div>5*</div><div>3</div><div>4</div></div>	<div><div>5*</div><div>3</div><div>4</div></div>
FIFO n+1=4	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>1*</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1*</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1*</div><div>2</div><div>3</div><div>4</div></div>	<div><div>5</div><div>2*</div><div>3</div><div>4</div></div>	<div><div>5</div><div>1</div><div>3*</div><div>4</div></div>	<div><div>5</div><div>1</div><div>2</div><div>4*</div></div>	<div><div>5*</div><div>1</div><div>2</div><div>3</div></div>	<div><div>4</div><div>1*</div><div>2</div><div>3</div></div>	<div><div>4</div><div>5</div><div>2*</div><div>3</div></div>



Using LRU





For LRU algorithm, the hit ratio always increases with the increase of cache block.

Time	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	1	2	3	4	1	2	5	1	2	3	4	5
LRU n=3 Hit 2 times	<div><div>1</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div></div>	<div><div>1*</div><div>2</div><div>3</div></div>	<div><div>4</div><div>2*</div><div>3</div></div>	<div><div>4</div><div>1</div><div>3*</div></div>	<div><div>4*</div><div>1</div><div>2</div></div>	<div><div>5</div><div>1*</div><div>2</div></div>	<div><div>5</div><div>1</div><div>2*</div></div>	<div><div>5*</div><div>1</div><div>2</div></div>	<div><div>3</div><div>1*</div><div>2</div></div>	<div><div>3</div><div>4</div><div>2*</div></div>	<div><div>3*</div><div>4</div><div>5</div></div>
LRU n=4 Hit 4 times	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>1*</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2*</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3*</div><div>4</div></div>	<div><div>1</div><div>2</div><div>5</div><div>4*</div></div>	<div><div>1</div><div>2</div><div>5</div><div>4*</div></div>	<div><div>1</div><div>2</div><div>5</div><div>4*</div></div>	<div><div>1</div><div>2</div><div>5*</div><div>3</div></div>	<div><div>1*</div><div>2</div><div>4</div><div>3</div></div>	<div><div>5</div><div>2*</div><div>4</div><div>3</div></div>
					Hit	Hit		Hit	Hit			



Belady

Time	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	1	2	3	4	1	2	5	1	2	3	4	5
FIFO n=3 Hit 3 times	1	1	1*	4	4	4*	5	5	5	5	5*	5*
		2	2	2*	1	1	1*	1*	1*	3	3	3
			3	3	3*	2	2	2	2	2*	4	4
								Hit	Hit			Hit
FIFO n+1=4 Hit 2 times	1	1	1	1*	1*	1*	5	5	5	5*	4	4
		2	2	2	2	2	2*	1	1	1	1*	5
			3	3	3	3	3	3*	2	2	2	2*
				4	4	4	4	4	4*	3	3	3
					Hit	Hit						



LRU

- How can I implement the LRU replacement algorithm with only ordinary gates and triggers?
- Comparison Pair Method
- Basic idea: Let each cache block be combined in pairs, use a ***comparison pair flip-flop*** to record the order in which the two cache blocks have been accessed in the comparison pair, and then use a gate circuit to combine the state of each comparison pair flip-flop, you can find the block to be replaced according to the LRU algorithm.



Example

- There are three cache blocks (A, B, and C), which can be combined into 6 pairs (AB, BA, AC, CA, BC, and CB). Among them, AB and BA, AC and CA, BC and CB are repeated, so only take AB, AC, BC.
- The access sequence of each pair is represented by “comparison pair flip-flops” T_{AB} , T_{AC} , and T_{BC} respectively. T_{AB} is "1", which means that A has been accessed more recently than B; T_{AB} is "0", which means that B has been accessed more recently than A. T_{AC} and T_{BC} are similarly defined.



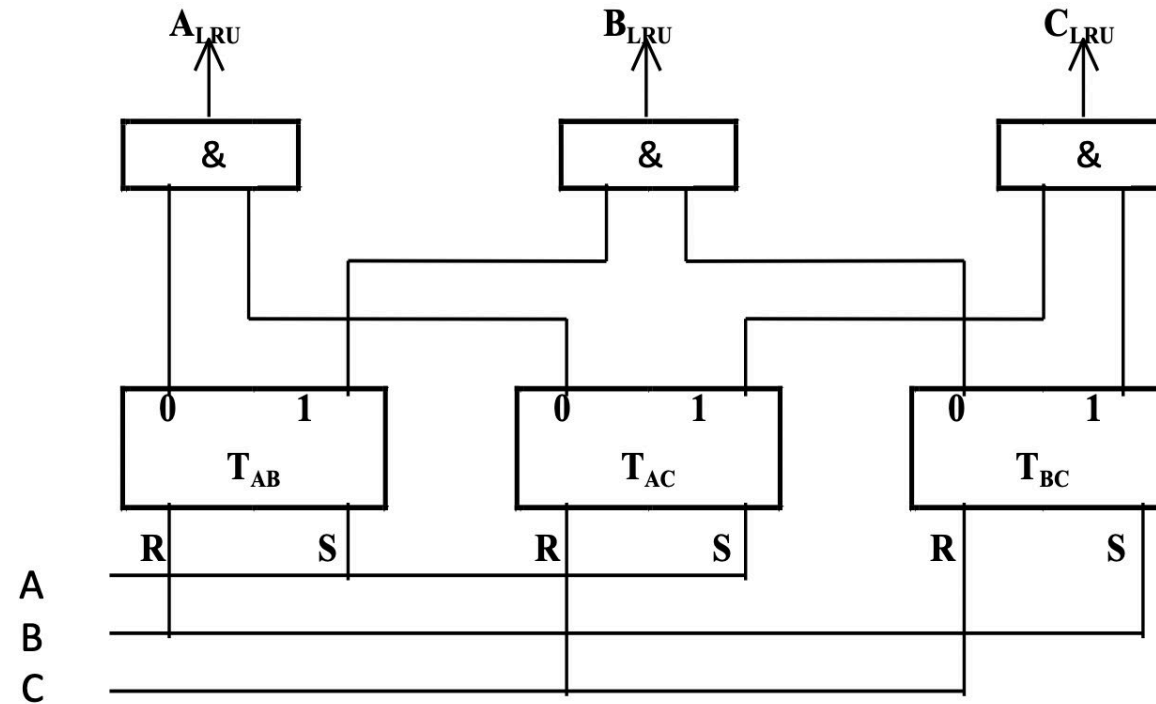
- If the most recently accessed block is A and C is the block that has not been accessed for the longest time, the three flip-flops' states must be respectively: $T_{AB}=1$, $T_{AC}=1$, and $T_{BC}=1$.
- If the most recently accessed block is B and C is the block that has not been accessed for the longest time, the three flip-flops' states must be respectively: $T_{AB}=0$, $T_{AC}=1$, and $T_{BC}=1$.
- Therefore, the block C that has not been accessed for the longest will be replaced. In that, the Boolean algebra expression must be:

$$C_{LRU} = T_{AB} \bullet T_{AC} \bullet T_{BC} + \overline{T_{AB}} \bullet T_{AC} \bullet T_{BC} = T_{AC} \bullet T_{BC}$$

$$B_{LRU} = T_{AB} \bullet \overline{T_{BC}}$$

$$A_{LRU} = \overline{T_{AB}} \bullet \overline{T_{AC}}$$





- Change the state of the flip-flop after each access.
 - After accessing block A: $T_{AB}=1$, $T_{AC}=1$
 - After accessing block B: $T_{AB}=0$, $T_{BC}=1$
 - After accessing block C: $T_{AC}=0$, $T_{BC}=0$

Hardware usage analysis (if p is the number of cache blocks)

- Since each block may be replaced, its signal needs to be generated with an AND gate, so the number of AND gates will be equal to p .
- Each AND gate receives inputs from its related flip-flops, for example, A_{LRU} AND gates must have inputs from T_{AB} and T_{AC} , B_{LRU} must have inputs from T_{AB} and T_{BC} , and the number of comparison pair flip-flops is the block number minus 1, so the input number of the AND gate is $p - 1$.
- If p is the block number, for pairwise combination, the number of comparison pair flip-flops should be C_p^2 , which is $p \cdot (p-1)/2$.



The Relationship between the Block Number and Required Hardware for Comparison Pair

Block Number	3	4	6	8	16	64	256
Number of Flip-flop	3	6	15	28	120	2016	32640
Number of And-Gate	3	4	6	8	16	64	256
Input Number of And-Gate	2	3	5	7	15	63	255



Q4: Write Strategy

- When data is written into the cache (on a store), is the data also written to main memory?
 - If the data is written to memory, the cache is called a **write-through cache**
 - Can always discard cached data - most up-to-date data is in memory
 - Cache control bit: only a valid bit
 - memory (or other processors) always have latest data
 - If the data is NOT written to memory, the cache is called a **write-back cache**
 - Can't just discard cached data - may have to write it back to memory
 - Cache control bits: both valid and dirty bits
 - much lower bandwidth, since data often overwritten multiple times
- Write-through adv: Read misses don't result in writes, memory hierarchy is consistent and it is simple to implement.
- Write back adv: Writes occur at speed of cache and main memory bandwidth is smaller when multiple writes occur to the same block.

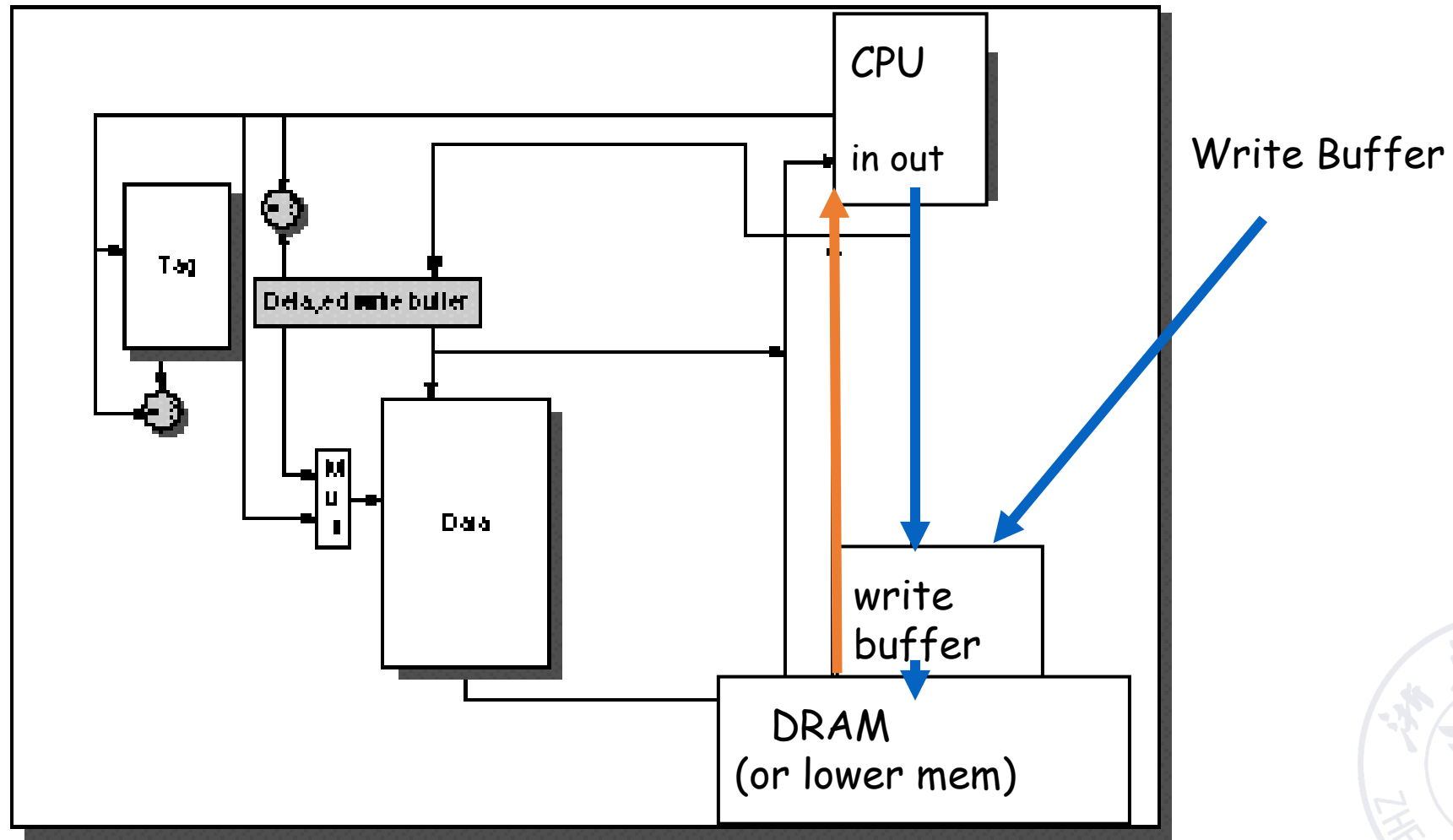


Write stall

- **Write stall** ---When the CPU must wait for writes to complete during write through
- Write buffers
 - A small cache that can hold a few values waiting to go to main memory.
 - To avoid stalling on writes, many CPUs use a write buffer.
 - This buffer helps when writes are clustered.
 - It does not entirely eliminate stalls since it is possible for the buffer to fill if the burst is larger than the buffer.



Write buffers



Write misses

- Write misses
 - If a miss occurs on a write (the block is not present), there are two options.
 - Write allocate
 - The block is loaded into the cache on a miss before anything else occurs.
 - Write around (no write allocate)
 - The block is only written to main memory
 - It is not stored in the cache.
- In general, write-back caches use write-allocate , and write-through caches use write-around .



Example

- Assume a fully associative write-back cache with many cache entries that starts empty below is a sequence of five memory operations(the address is in square brackets):

```
1  write Mem[100];  
2  write Mem[100];  
3  read  Mem[200];  
4  write Mem[200];  
5  write Mem[100];
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Answer :

for no-write allocate	misses: 1,2,3,5	hit: 4
for write allocate	misses: 1,3	hit: 2,4,5



How memory hierarchy works?

Questions:

- Q1. *Where can a block be placed in main memory?*
- Q2. *How is a block found if it is in main memory?*
- Q3. *Which block should be replaced on a virtual memory miss?*
- Q4. *What happens on a write?*

Memory System Performance

CPU Execution time

- CPU Execution time = (CPU clock cycles + Memory stall cycles) × Clock cycle time

Memory stall cycles = IC × MemAccess refs per instructions × Miss rate × Miss penalty

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

- CPI Execution includes ALU and Memory instructions



Average Memory Access Time

- Average Memory Access Time

$$\begin{aligned}
 \text{Average Memory Access Time} &= \frac{\text{Whole accesses time}}{\text{All memory accesses in program}} \\
 &= \frac{\text{Accesses time on hitting} + \text{Accesses time on miss}}{\text{All memory accesses in program}} \\
 &= \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty}) \\
 &= (\text{HitTime}_{Inst} + \text{MissRate}_{Inst} \times \text{MissPenalty}_{Inst}) \times \text{Inst}\% \\
 &\quad + (\text{HitTime}_{Data} + \text{MissRate}_{Data} \times \text{MissPenalty}_{Data}) \times \text{Data}\%
 \end{aligned}$$

$$\text{CPUtime} = IC \times \left(\frac{\text{AluOps}}{\text{Inst}} \times \text{CPI}_{\text{AluOps}} + \frac{\text{MemAccess}}{\text{Inst}} \times \text{AMAT} \right) \times \text{CycleTime}$$



Example1: Impact on Performance

- Suppose a processor executes at
 - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
 - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty
- What is the CPUtime and the AMAT ?

• Answer: CPI = ideal CPI + average stalls per instruction = $1.1(\text{cycles/ins}) + [0.30(\text{DataMops/ins}) \times 0.10(\text{miss/DataMop}) \times 50(\text{cycle/miss})] + [1(\text{InstMop/ins}) \times 0.01(\text{miss/InstMop}) \times 50(\text{cycle/miss})] = (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1$

• AMAT = $(1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2.54$



Example2: Impact on Performance

Assume : Ideal CPI=1 (no misses)

- L/S's structure . 50% of instructions are data accesses
- Miss penalty is 25 clock cycles
- Miss rate is 2%
- How faster would the computer be if all instructions were cache hits?

Answer: first compute the performance for always hits:

$$\begin{aligned}
 \text{CPU}_{\text{execution time}} &= (\text{CPU clock cycles} + \text{memory stall cycles}) \times \text{clock cycle} \\
 &= (IC \times \text{CPI} + 0) \times \text{Clock cycle} \\
 &= IC \times 1.0 \times \text{clock cycle}
 \end{aligned}$$

- Now for the computer with the real cache, first compute memory stall cycles:

$$\begin{aligned}
 \text{Memory stall cycles} &= IC \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Missrate} \times \text{Miss penalty} \\
 &= IC \times (1 + 0.5) \times 0.02 \times 25 = IC \times 0.75
 \end{aligned}$$



Example2: Impact on Performance

Assume: Ideal CPI=1 (no misses)

- L/S's structure . 50% of instructions are data accesses
- Miss penalty is 25 clock cycles
- Miss rate is 2%
- How faster would the computer be if all instructions were cache hits?

The total performance is thus:

$$\begin{aligned}\text{CPU execution time cache} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle} \\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$



Example2: Impact on Performance

Assume: Ideal CPI=1 (no misses)

- L/S's structure . 50% of instructions are data accesses
- Miss penalty is 25 clock cycles
- Miss rate is 2%
- How faster would the computer be if all instructions were cache hits?

The performance ratio is the inverse of the execution times

$$\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} = \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{clock cycle}} = 1.75$$

The computer with no cache misses is 1.75 time faster.



Example3: Impact on Performance

Assume: CPI=2(perfect cache) clock cycle time = 1.0 ns

- MPI(memory reference per instruction) = 1.5
- Size of both caches is 64K and size of both block is 64 bytes
- One cache is direct mapped and other is two-way set associative. the former has miss rate of 1.4%, the latter has miss rate 1.0%
- The selection multiplexor forces CPU clock cycle time to be stretched 1.25 times
- Miss penalty is 75ns, and hit time is 1 clock cycle
- What is the impact of two cache organizations on performance of CPU (first, calculate the average memory access time and then CPU performance)?

Answer :

Average memory access time is

- Average memory access time = Hit time + Miss rate \times miss penalty

Thus, the time for each organization is

- Average memory access time 1-way = $1.0 + (0.014 \times 75) = 2.05$ ns
- Average memory access time 2-way = $1.0 \times 1.25 + (0.01 \times 75) = 2.00$ ns



Example3: Impact on Performance

The average memory access time is better for the 2-way set-associative cache.

CPU performance is

$$\begin{aligned}
 CPUtime &= IC \times \left(CPI_{execution} + \frac{Misses}{Instruction} \times Miss\ penalty \right) \times Clock\ cycle\ time \\
 &= IC \times \left[\left(CPI_{execution} \times Clock\ cycle\ time \right) \right. \\
 &\quad \left. + \left(Miss\ rate \times \frac{Memory\ accesses}{Instruction} \times Miss\ penalty \times Clock\ cycle\ time \right) \right]
 \end{aligned}$$

Substituting 75 ns for (miss penalty \times Clock cycle time), the performance of each cache organization is

$$CPU\ time_{1-way} = IC \times (2 \times 1.0 + (1.5 \times 0.014 \times 75)) = 3.58 \times IC$$

$$CPU\ time_{2-way} = IC \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.010 \times 75)) = 3.63 \times IC$$



Example3: Impact on Performance

Relative performance is

$$\frac{CPU\ time_{2-way}}{CPU\ time_{1-way}} = \frac{3.63 \times Instruction\ count}{3.58 \times Instruction\ count} = \frac{3.63}{3.58} = 1.01$$

In contrast to the results of average memory access time, the direct-mapped leads to slightly better average performance. Since CPU time is our bottom-line evaluation.



How to Improve

Hence, there are more than 20 cache optimizations into these categories:

$$AMAT = HitTime + MissRate \times MissPenalty$$

1. Reduce the miss penalty
 - multilevel caches, critical word first, read miss before write miss, merging write buffers, and victim caches
2. Reduce the miss rate
 - larger block size, large cache size, higher associativity, way prediction and pseudo-associativity, and compiler optimizations
3. Reduce the time to hit in the cache
 - small and simple caches, avoiding address translation, pipelined cache access, and trace caches³.
4. Reduce the miss penalty and miss rate via parallelism
 - non-blocking caches, hardware prefetching, and compiler prefetching



Summary of Cache Optimization Technology

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size \neq L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

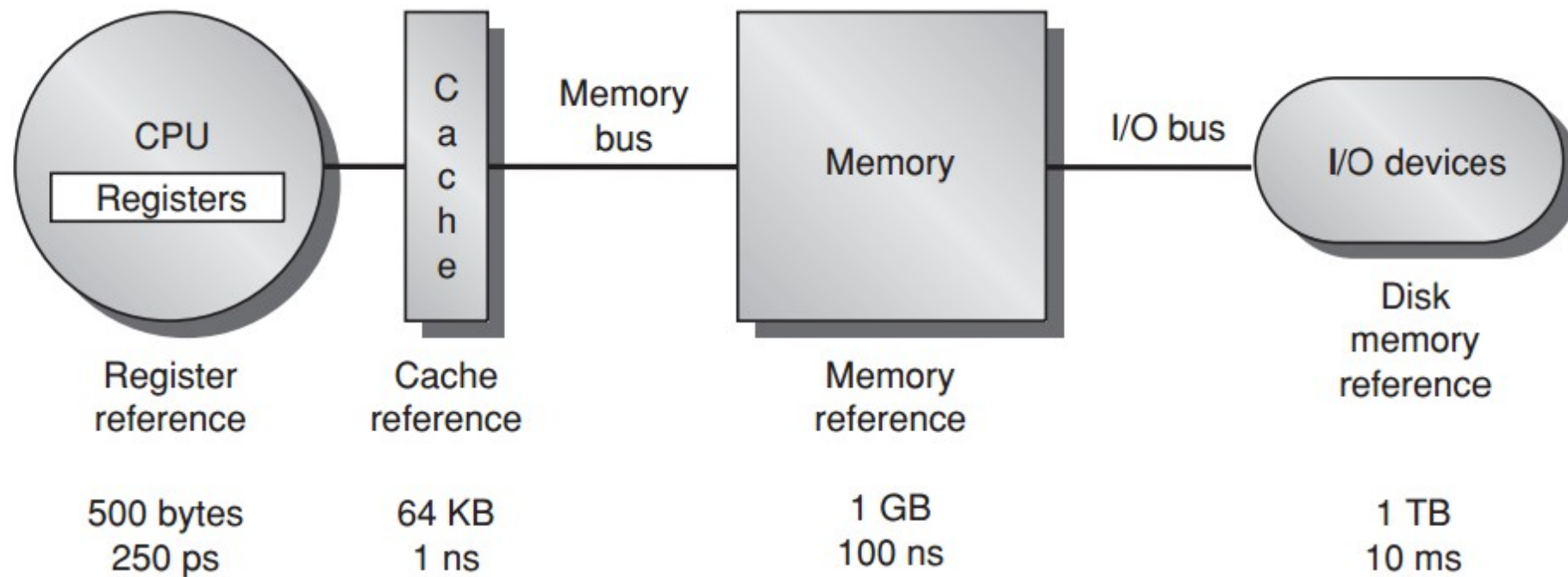


Summary of Cache Optimization Technology

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache		+/-	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

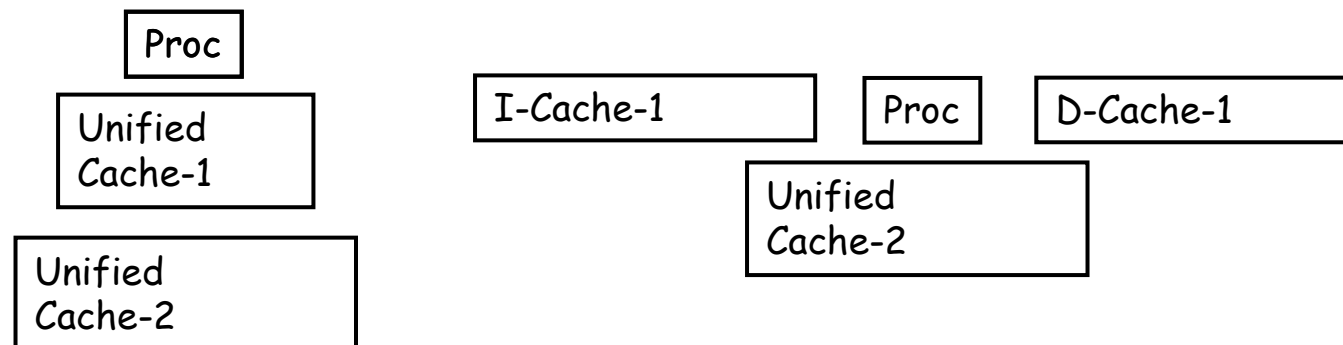


Memory Hierarchy

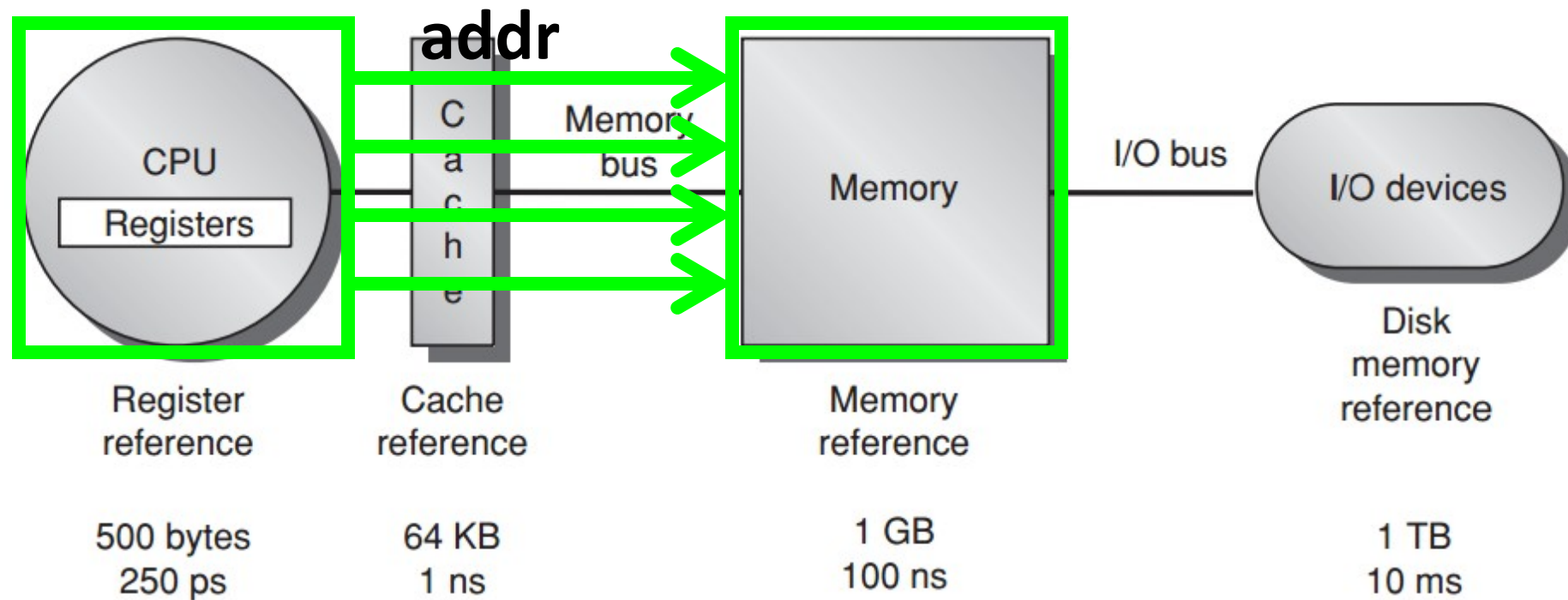


Split vs. unified caches

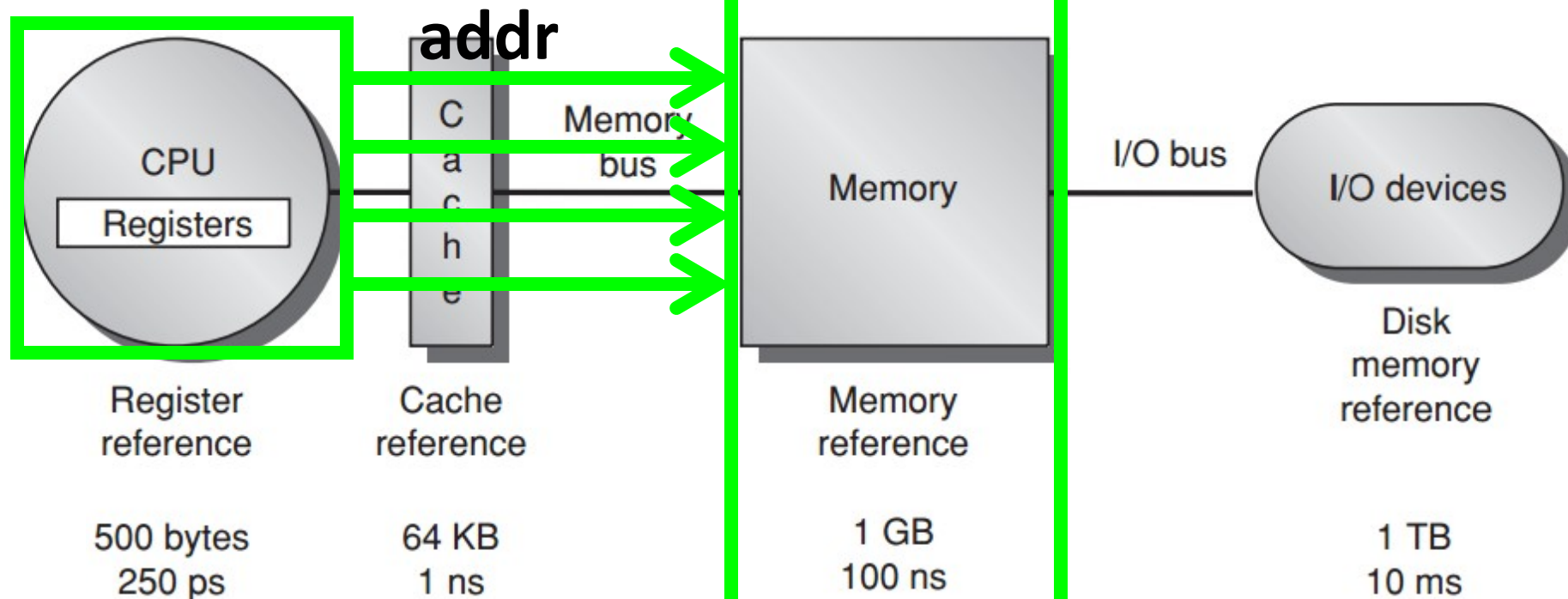
- Unified cache
 - All memory requests go through a single cache.
 - This requires less hardware, but also has lower performance
- Split I & D cache
 - A separate cache is used for instructions and data.
 - This uses additional hardware, though there are some simplifications (the I cache is read-only).



Memory Hierarchy

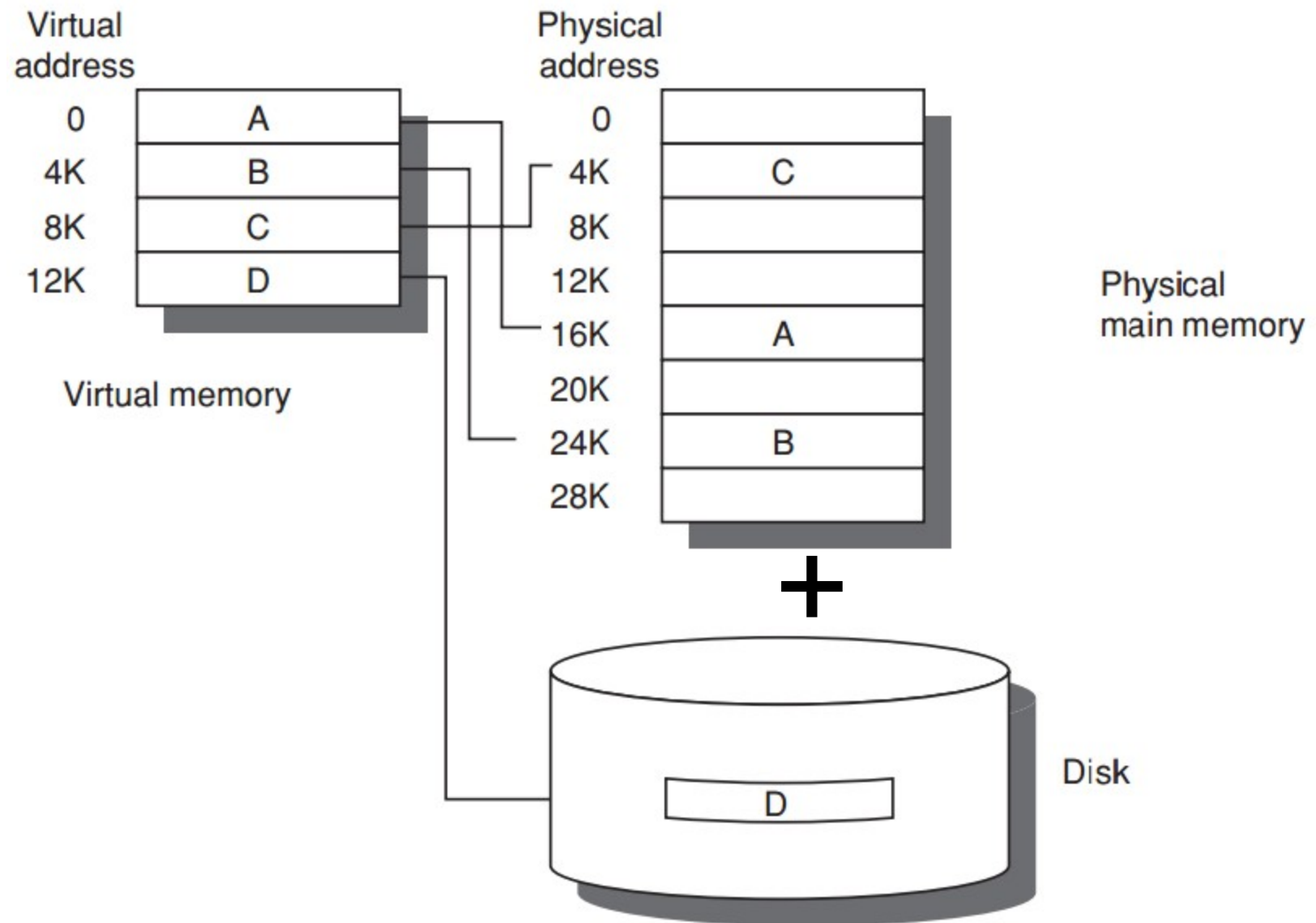


Memory Hierarchy

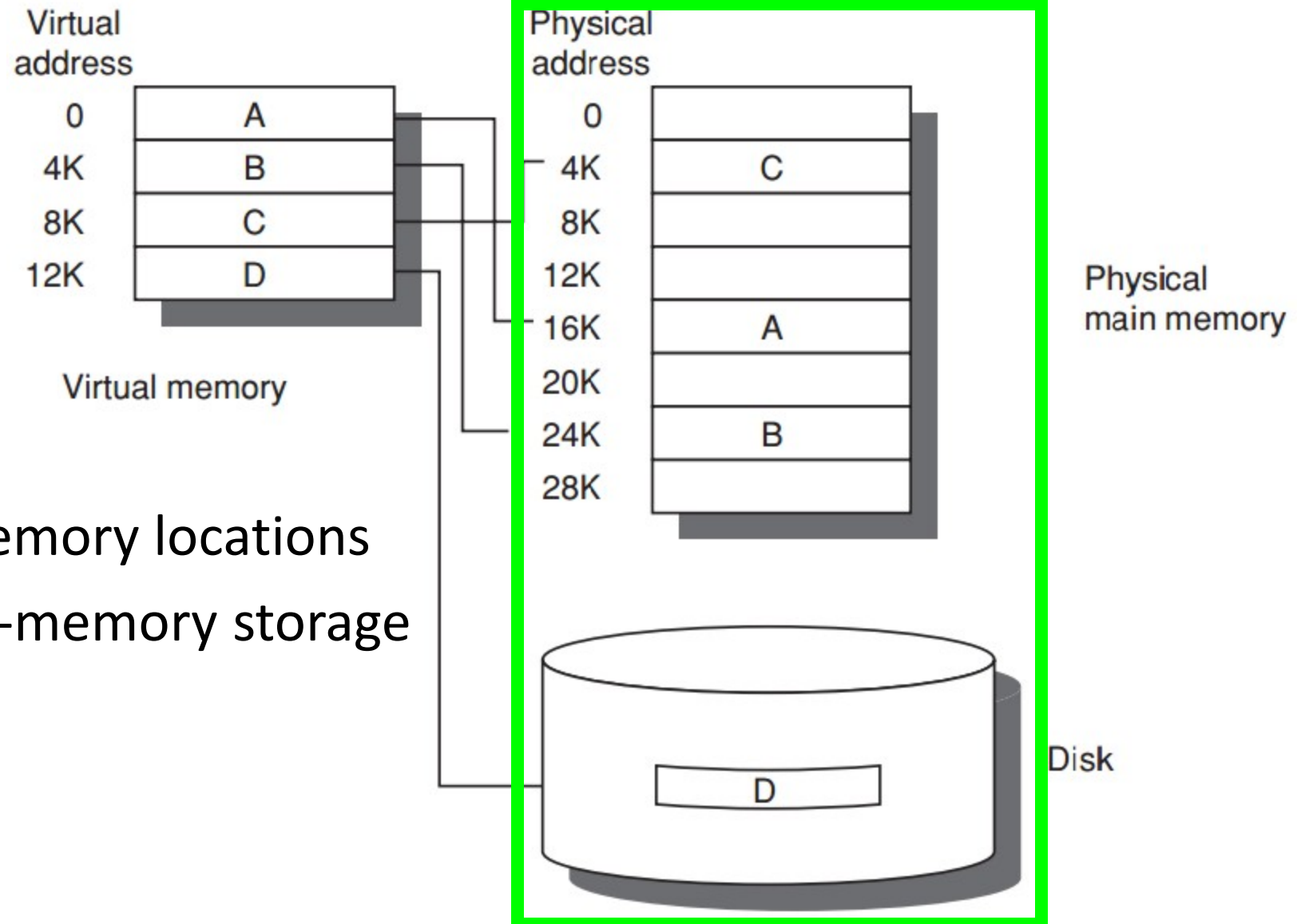


Larger memory for more processes?

Virtual Memory



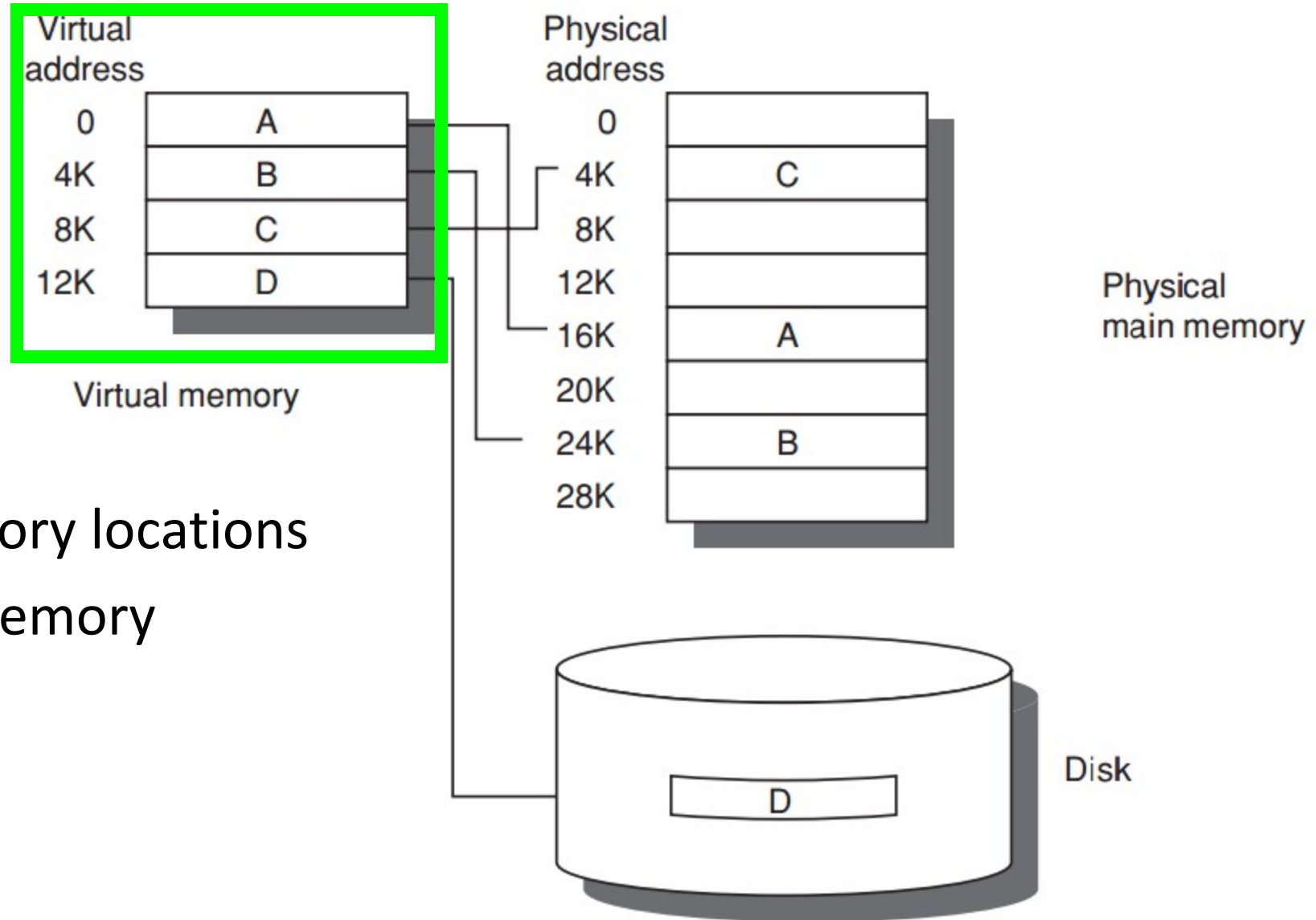
Virtual Memory



Program **uses**

- discontinuous memory locations
- + secondary/non-memory storage

Virtual Memory



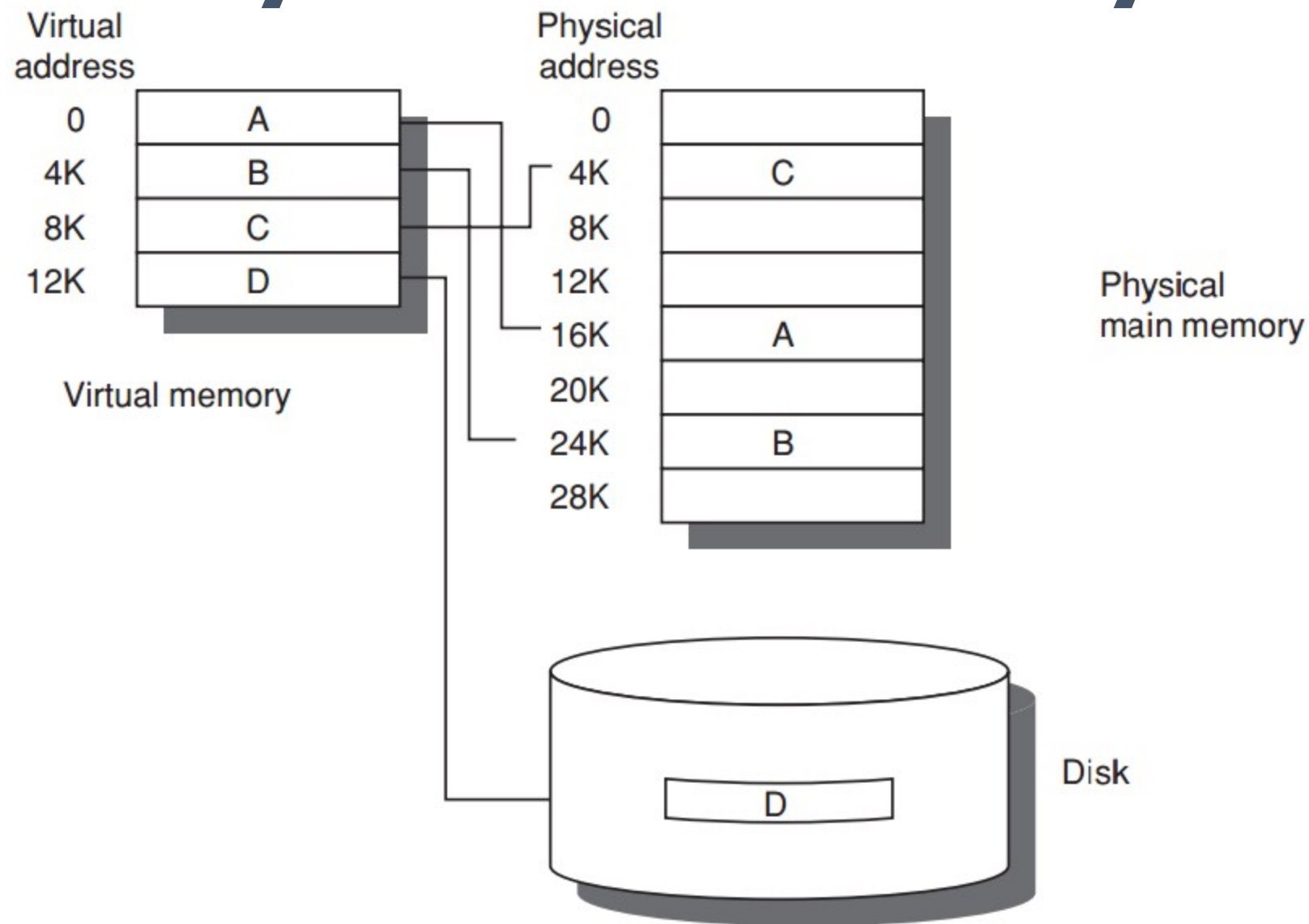
Program **thinks**

- contiguous memory locations
- larger physical memory

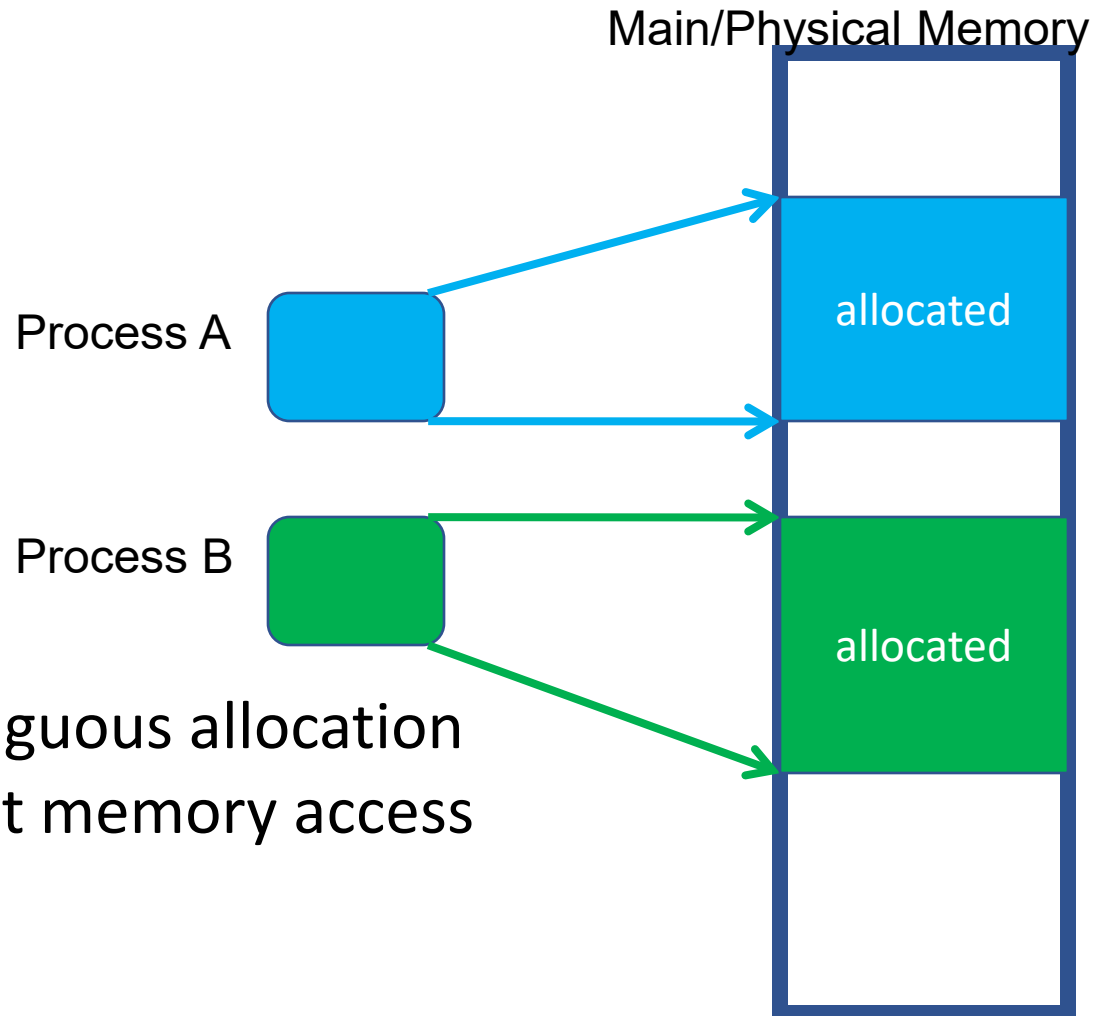
Preview

- Why virtual memory (besides larger)?
- Virtual-physical address translation?
- Memory protection/sharing among multi-program?

more behind the scenes: why virtual memory?

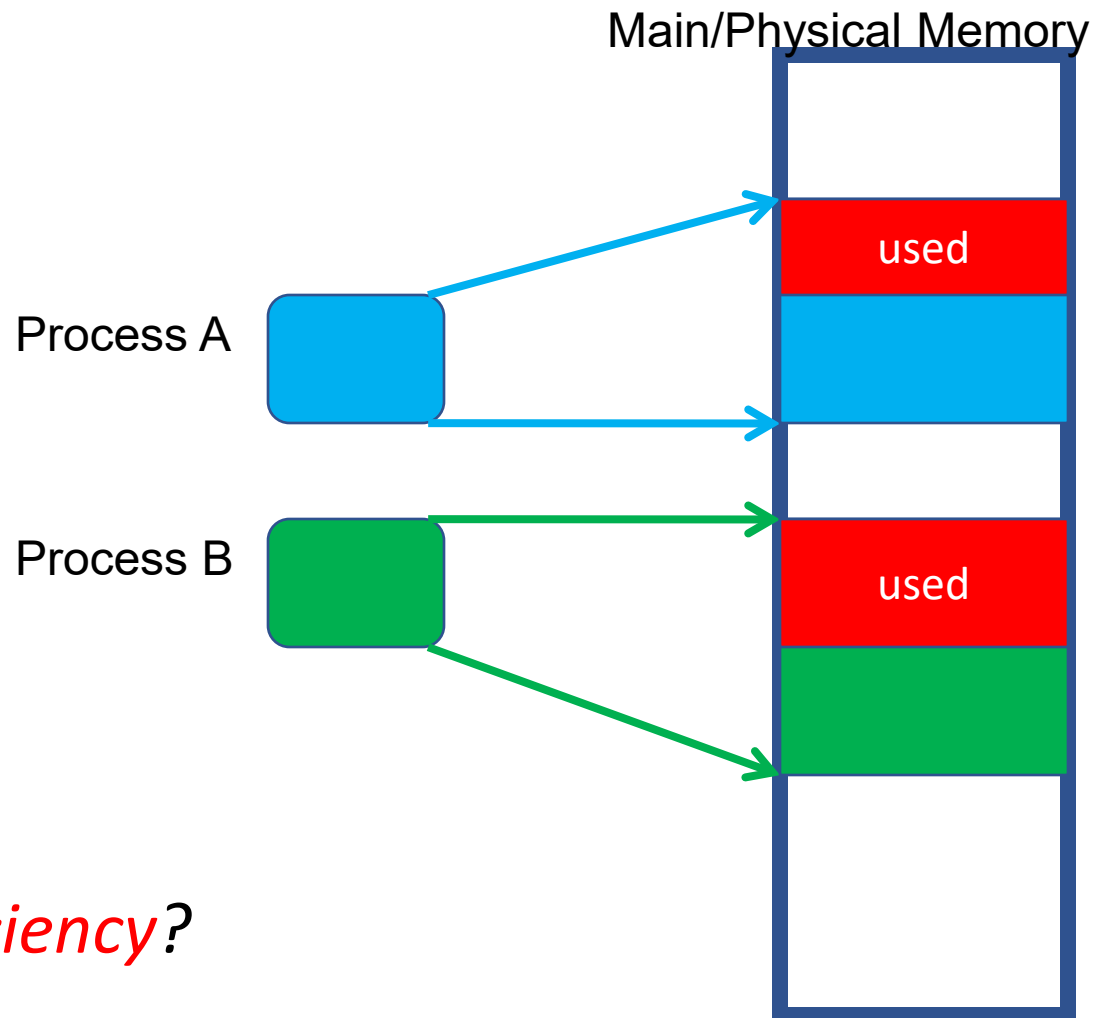


Prior Virtual Memory



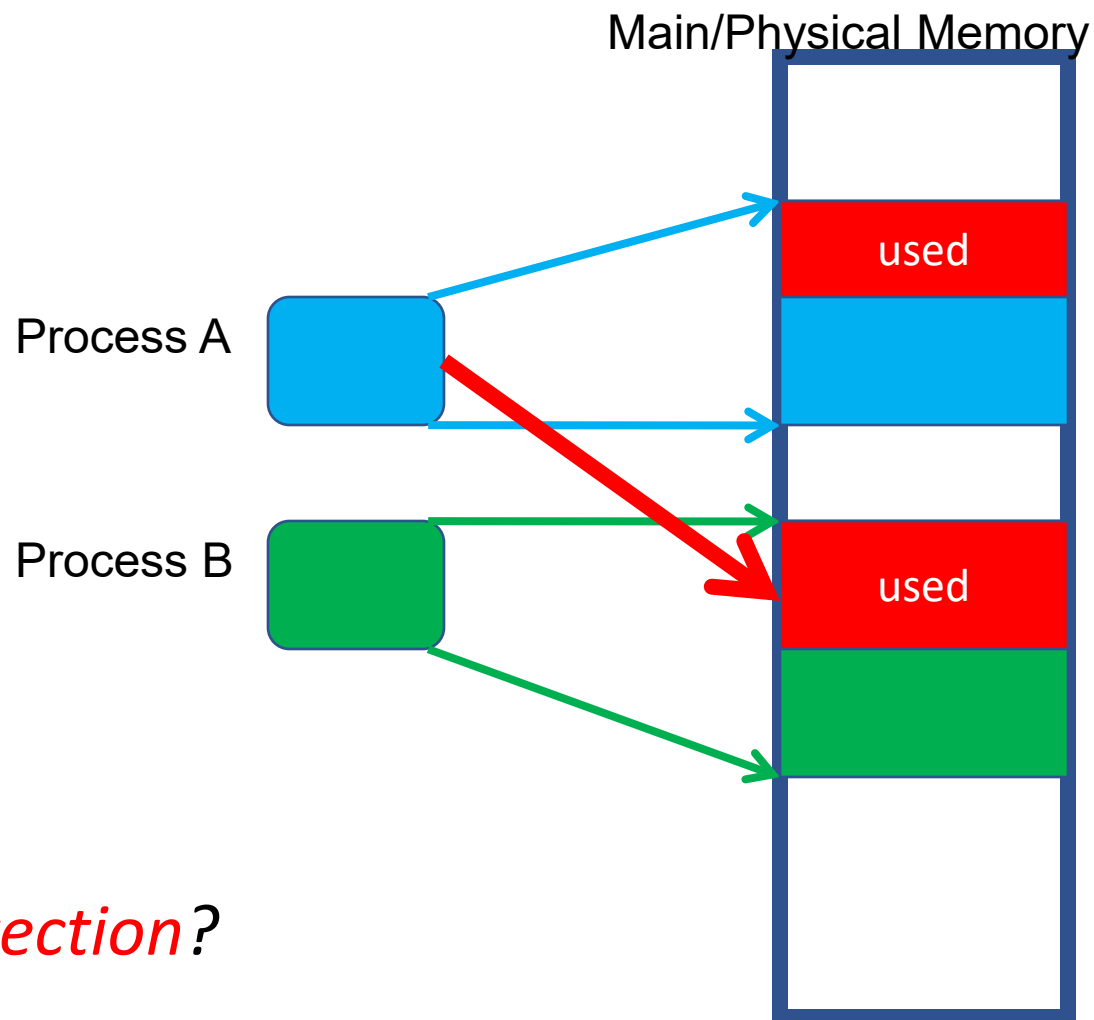
- Contiguous allocation
- Direct memory access

Prior Virtual Memory



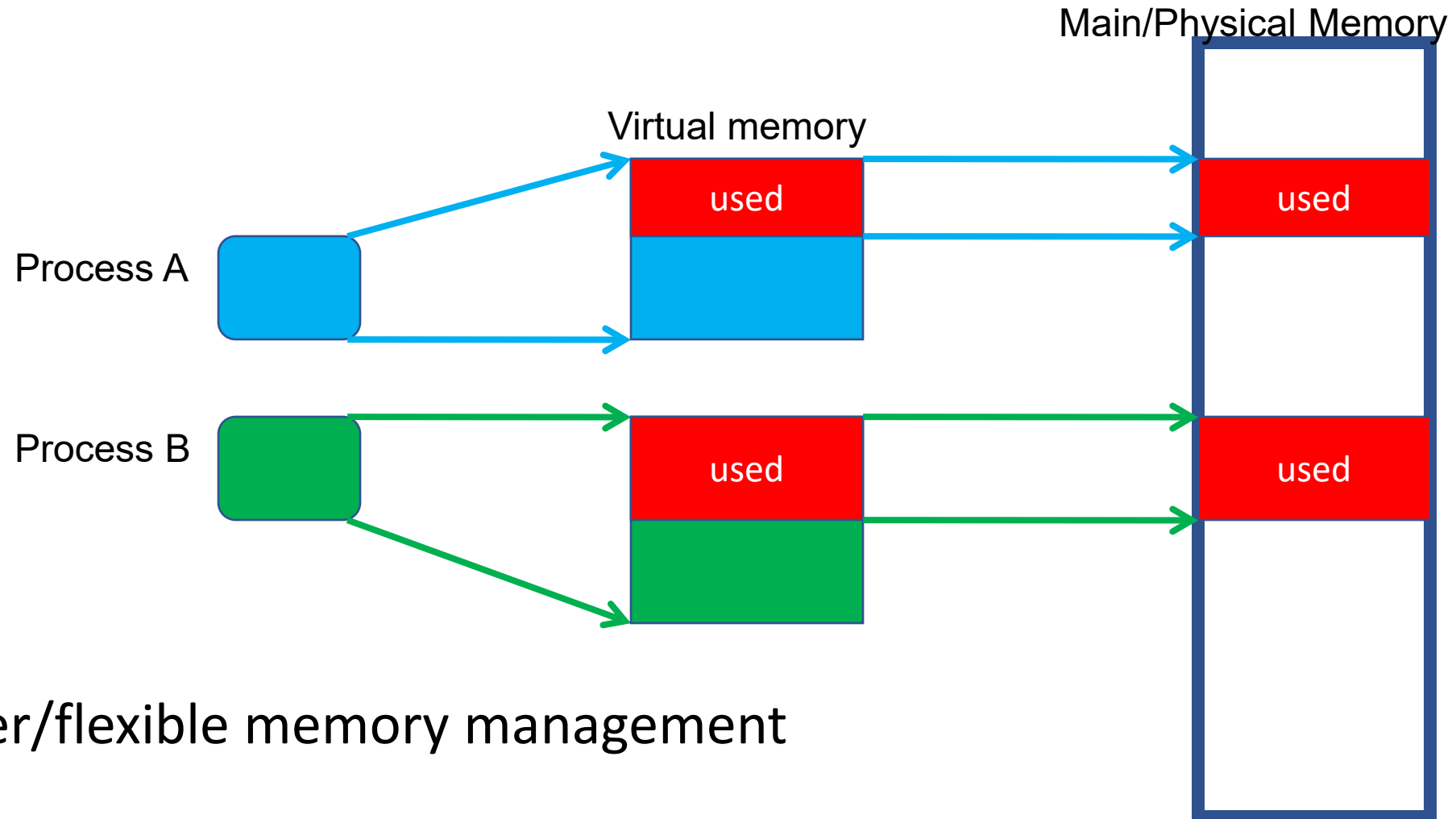
- *efficiency?*

Prior Virtual Memory



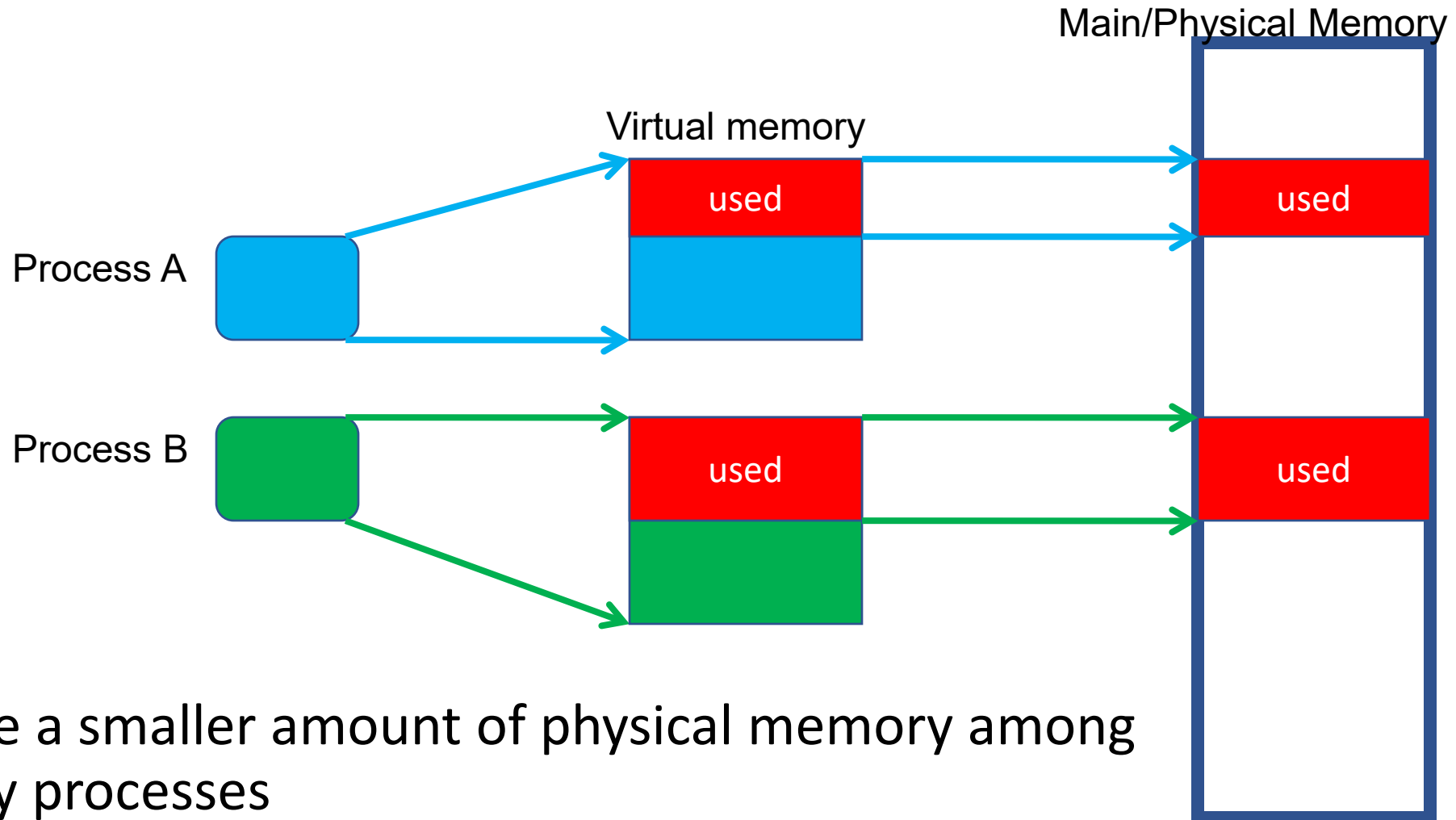
- *protection?*

Virtual Memory



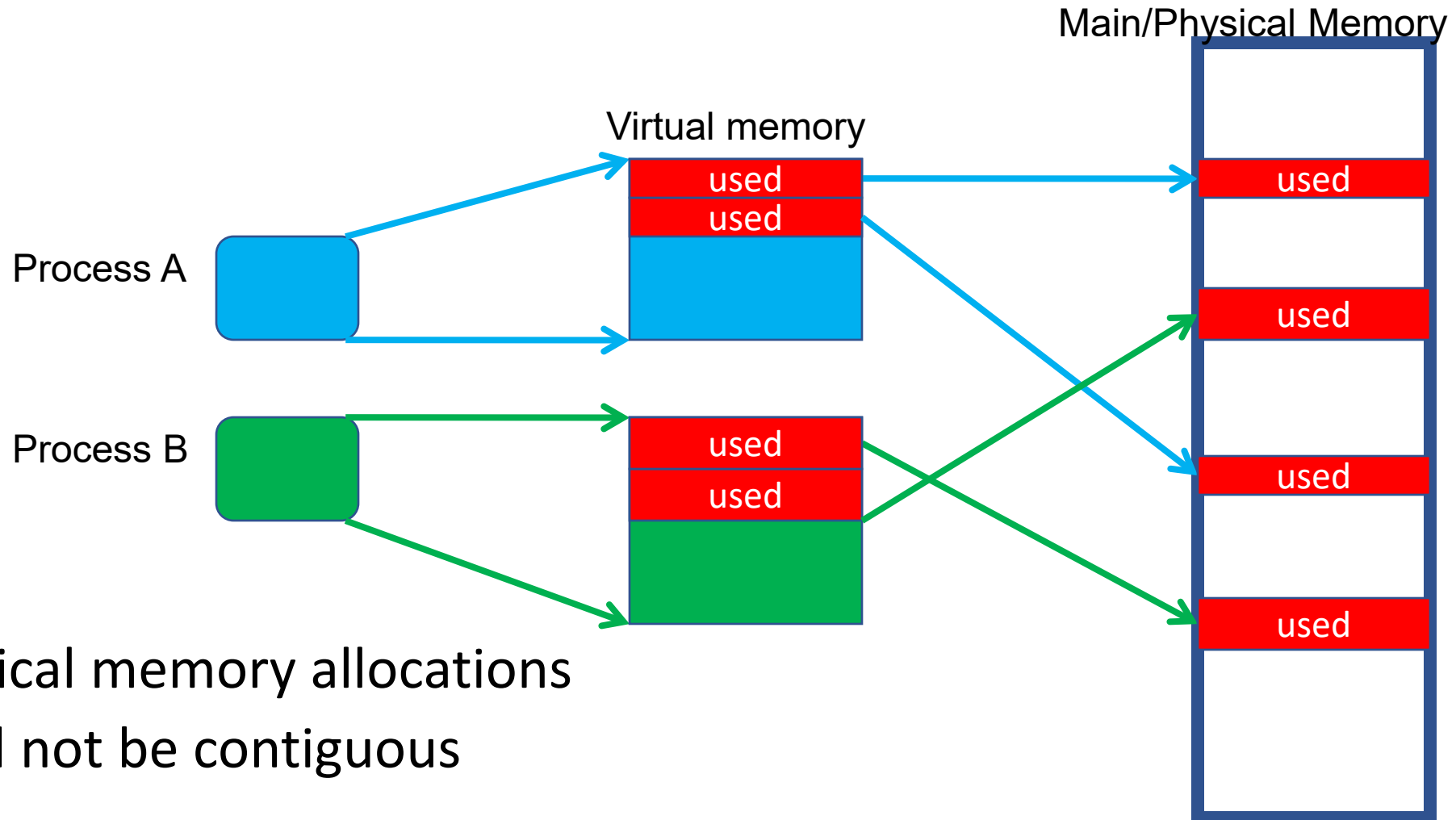
- Easier/flexible memory management

Virtual Memory



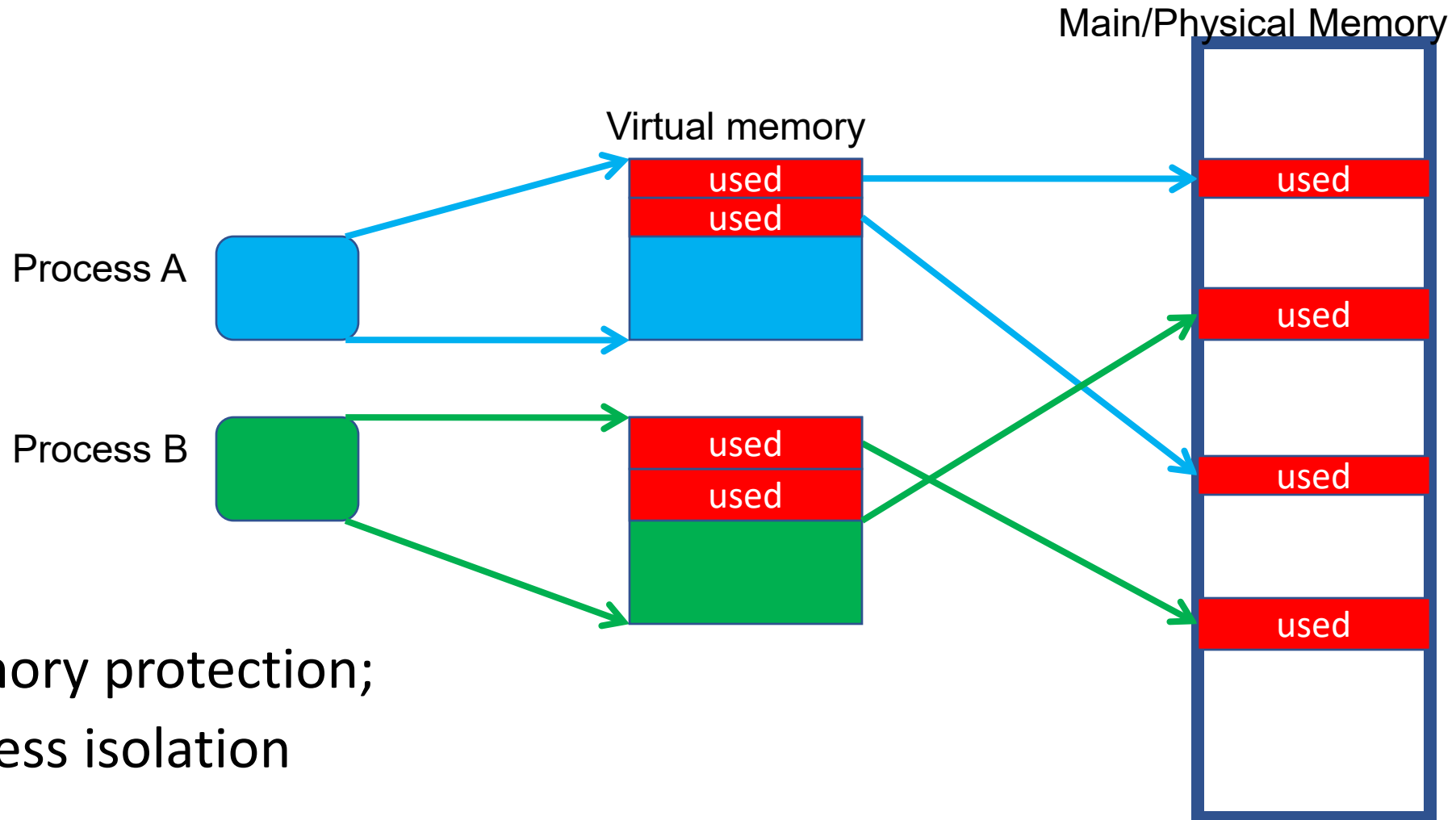
- Share a smaller amount of physical memory among many processes

Virtual Memory

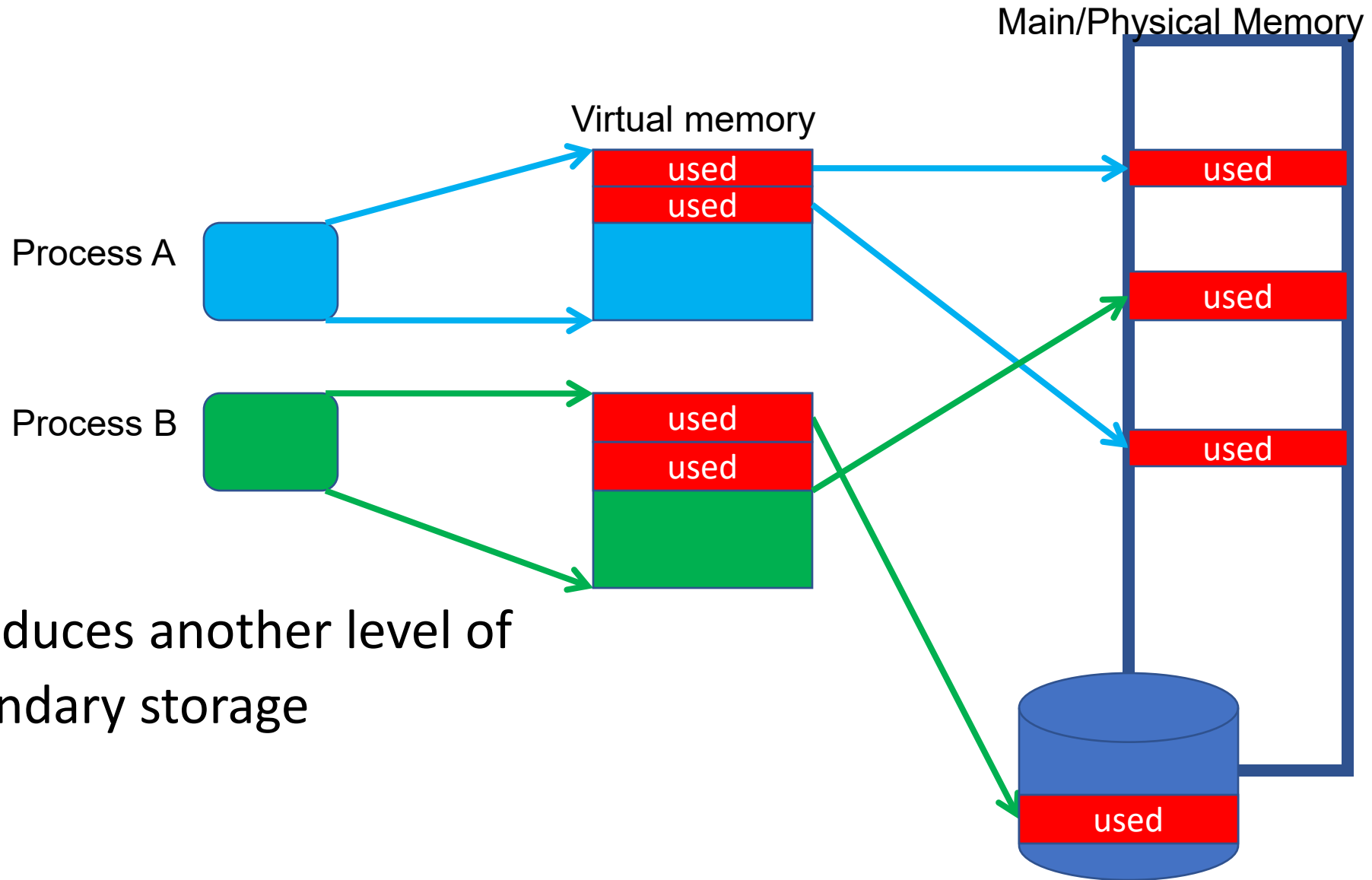


- Physical memory allocations need not be contiguous

Virtual Memory

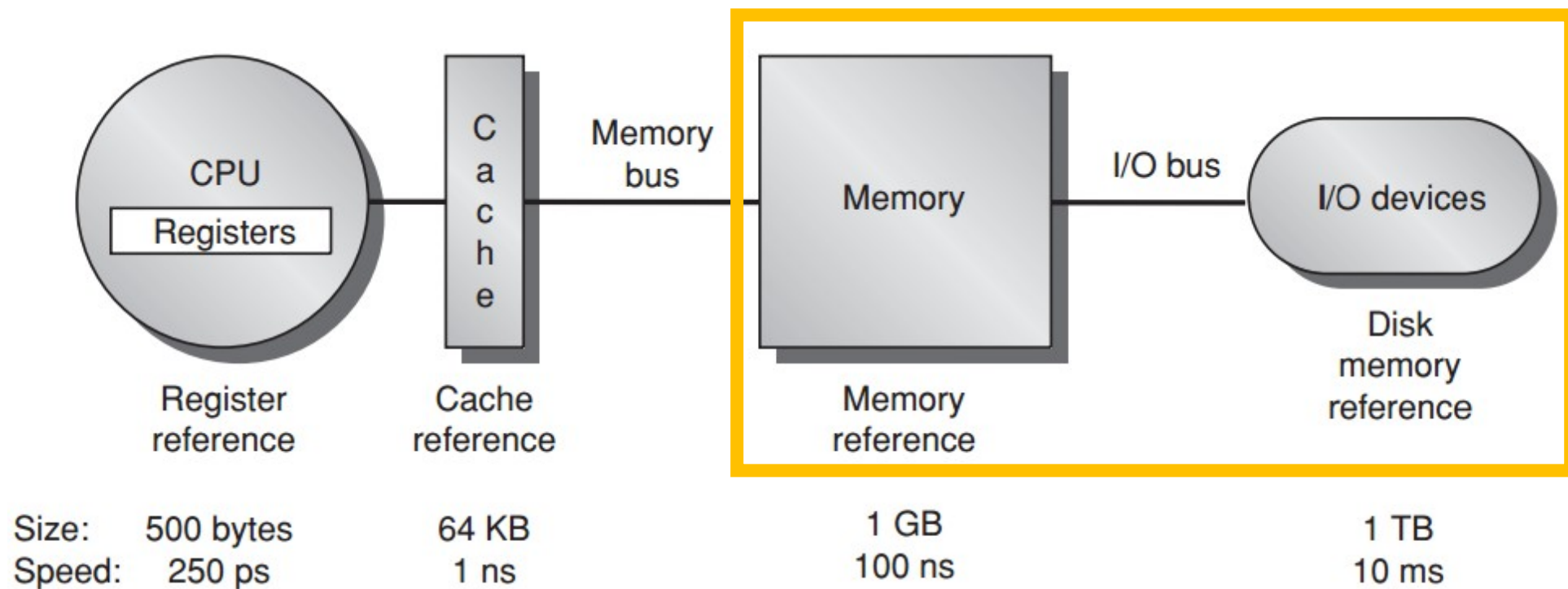


Virtual Memory



- Introduces another level of secondary storage

Virtual Memory = Main Memory + Secondary Storage

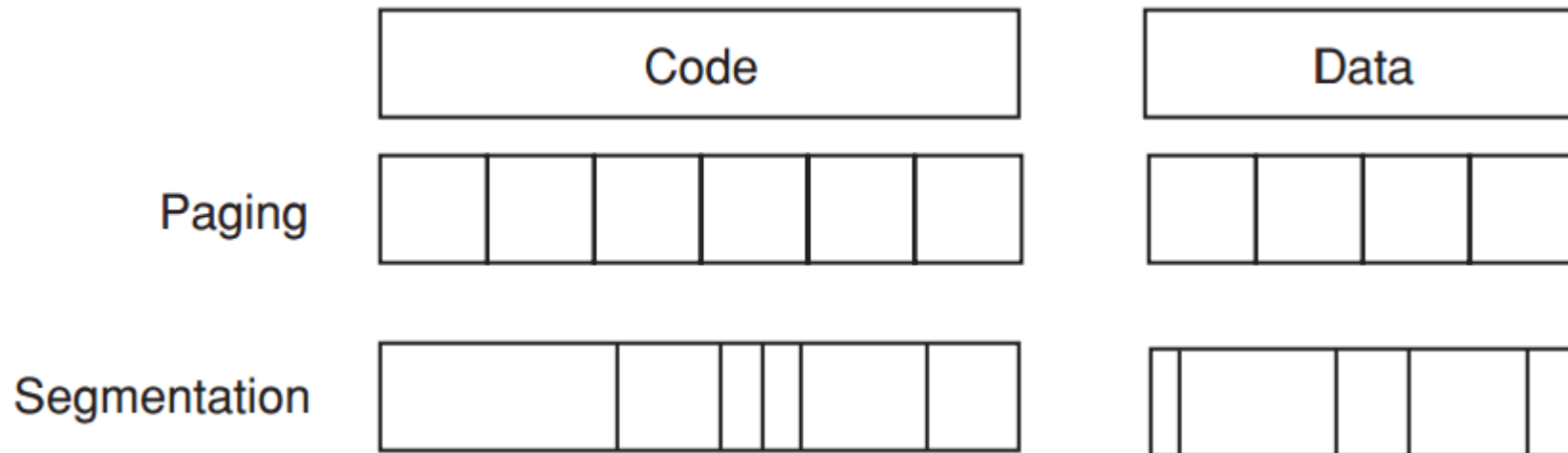


Cache vs Virtual Memory

Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	100–200 clock cycles
Miss penalty	8–200 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–160 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45 bit physical address to 14–20 bit cache address	32–64 bit virtual address to 25–45 bit physical address

Virtual Memory Allocation

- Paged virtual memory
page: fixed-size block
- Segmented virtual memory
segment: variable-size block



Virtual Memory

Address

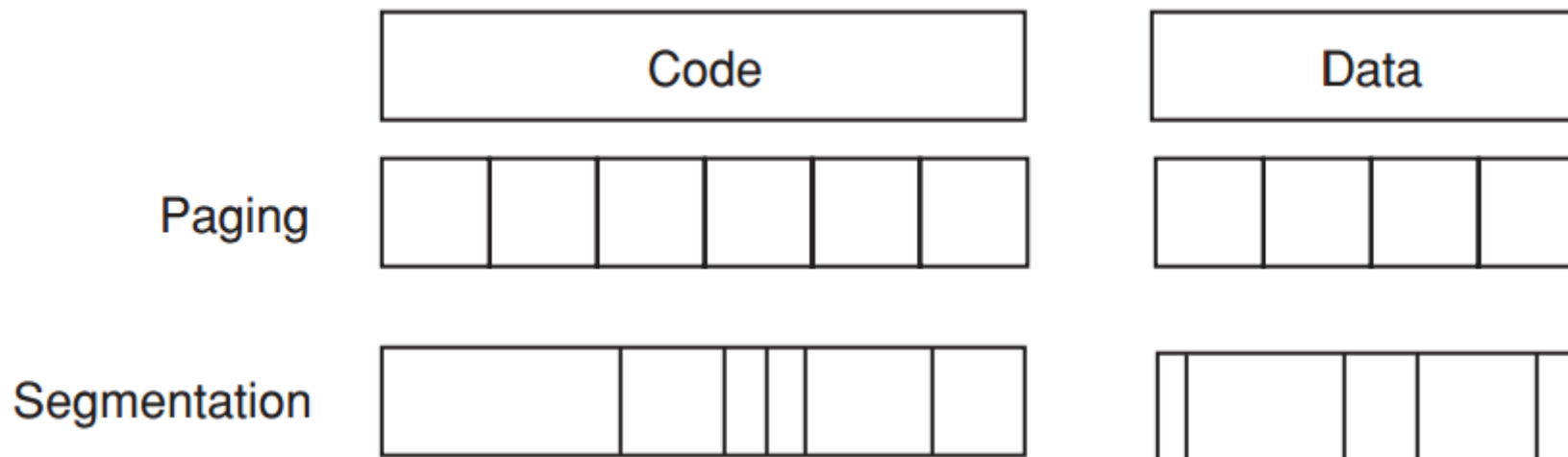
- Paged virtual memory

page address: page # || offset

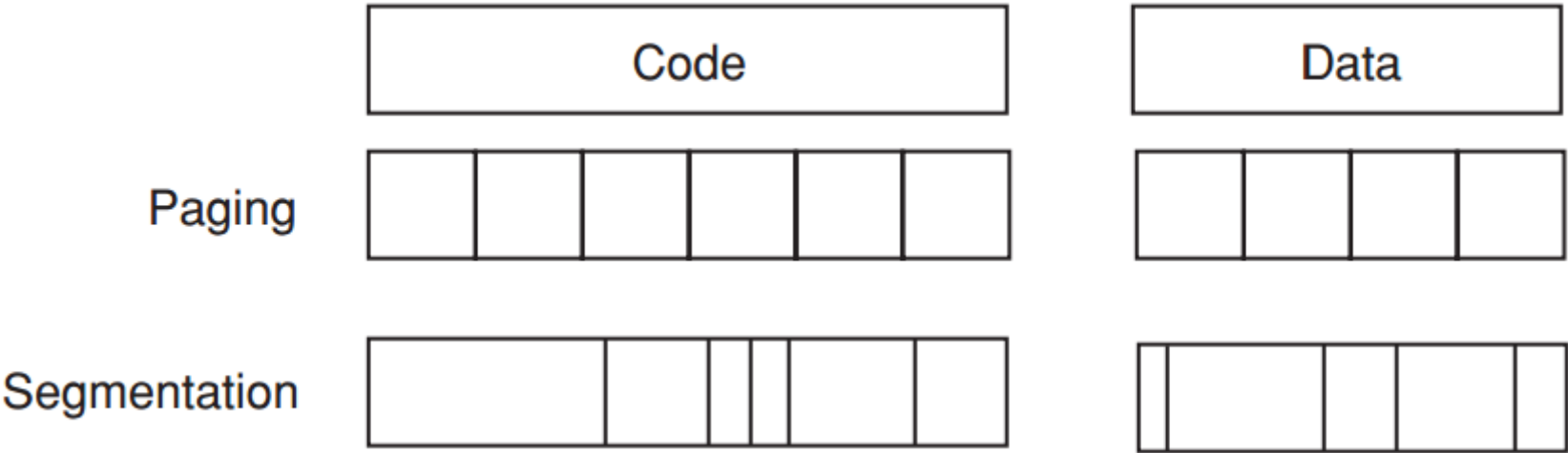
one word
- Segmented virtual memory

segment address: seg # | offset

two words



Pros & Cons?



Paging vs Segmentation

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find <u>contiguous, variable-size, unused portion of main memory</u>) http://www.cnblogs.com/felixfang/p/3420462.html
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

How virtual memory works?

How virtual memory works?

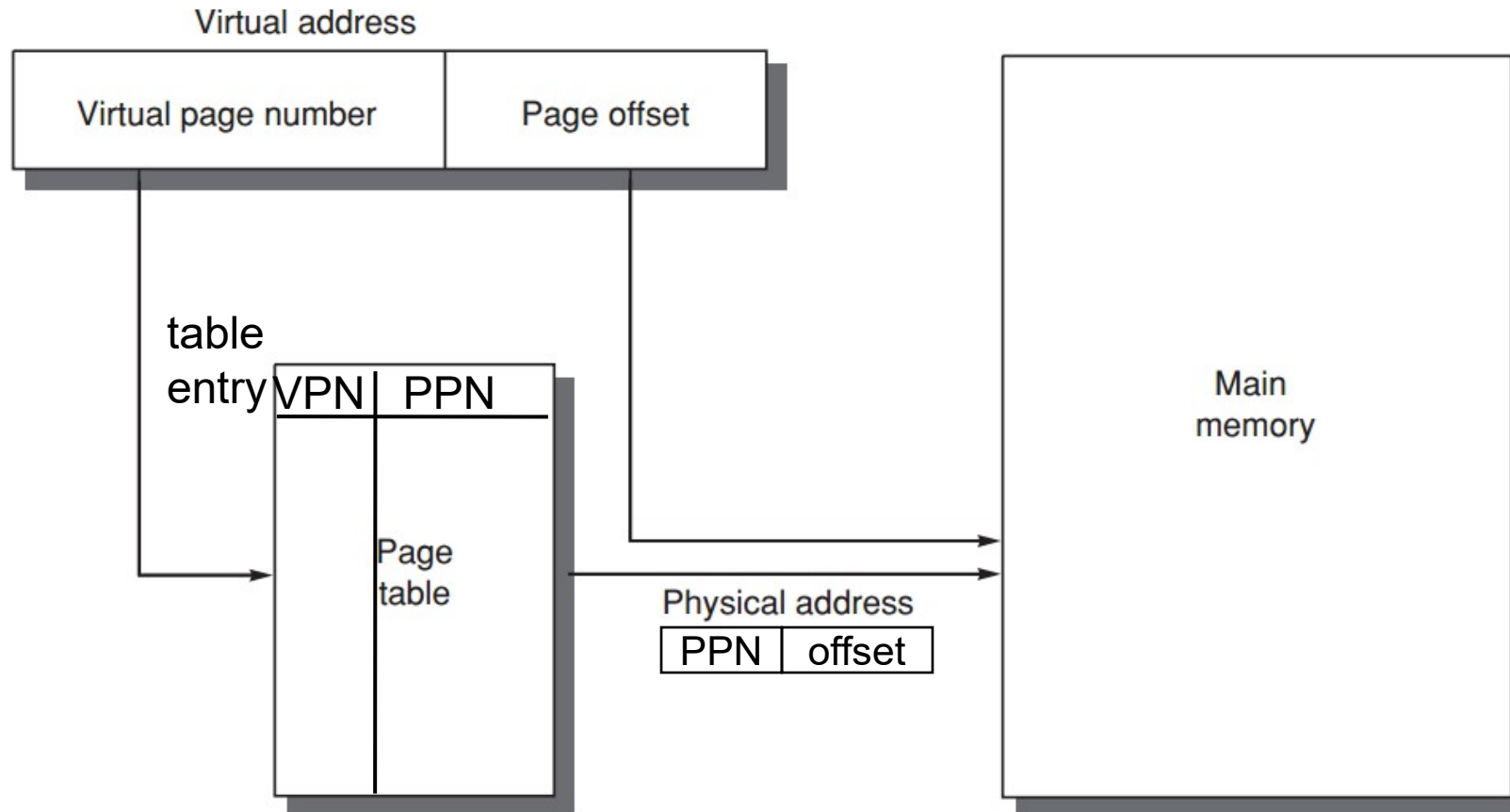
Four Questions

Four Mem Hierarchy Q's

- **Q1. *Where can a block be placed in main memory?***
- Fully associative strategy:
OS allows blocks to be placed anywhere in main memory
- Because of high miss penalty by
access to a rotating magnetic storage device upon page/address fault

Four Mem Hierarchy Q's

- ***Q2. How is a block found if it is in main memory?***



Four Memory Hierarchy Q's

- **Q3. *Which block should be replaced on a virtual memory miss?***
- Least recently used (LRU) block
- use/reference bit
 - logically set whenever a page is accessed;
 - OS periodically clears use bits and later records them to track the least recently referenced pages;

Four Mem Hierarchy Q's

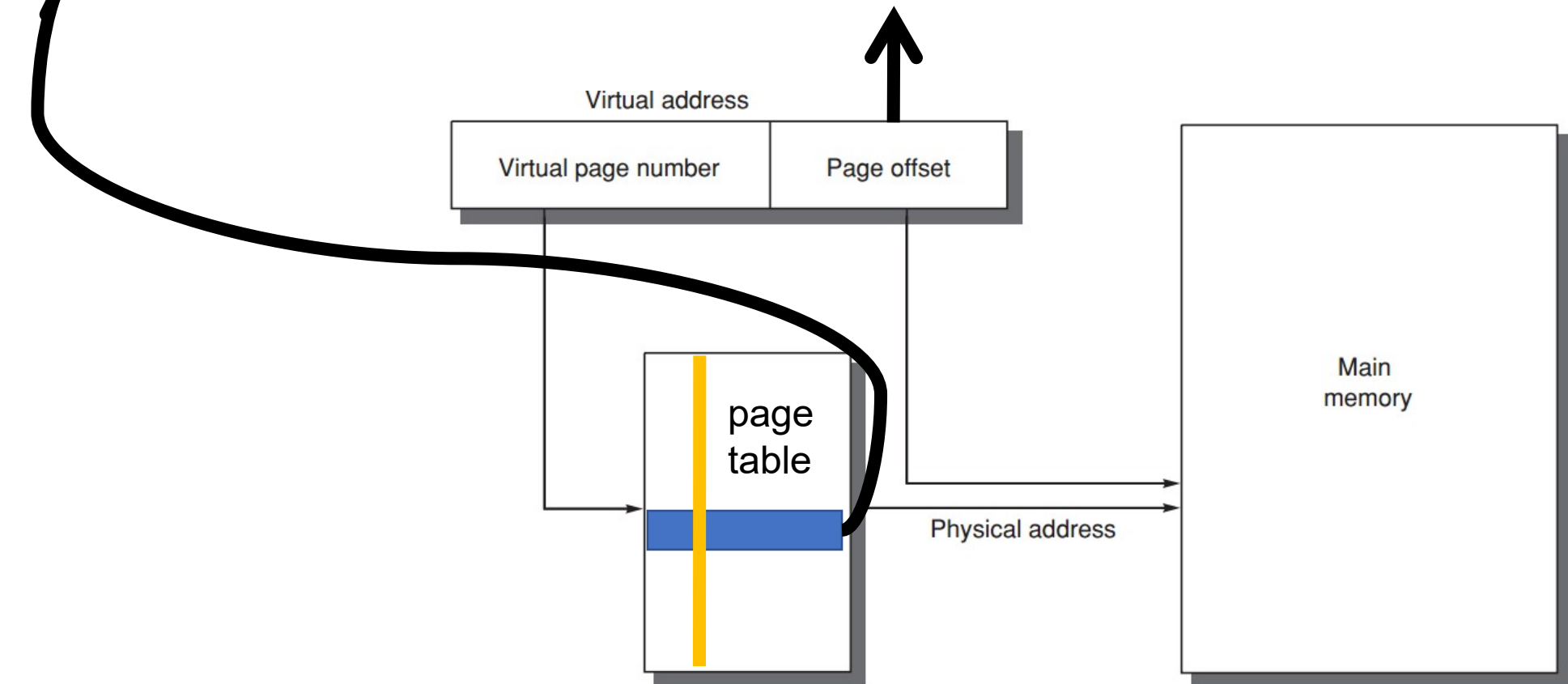
- **Q4. *What happens on a write?***
- Write-back strategy
 - as accessing rotating magnetic disk takes millions of clock cycles;
- Dirty bit
 - write a block to disk only if it has been altered since being read from the disk;

well...tell me more

well...tell me more
Address Translation

Page Table

physical page number || page offset = physical address



Page Table ?

- Page tables are often large

32-bit virtual address, 4KB pages,
4 bytes per page table entry.

page table size:

$$(2^{32}/2^{12}) \times 2^2 = 2^{22} \text{ bytes} = 4 \text{ MB}$$

Page Table ?

- Page tables are stored in main memory
- Logically two memory accesses for data access:
 - one to obtain the physical address from page table;
 - one to get the data from the physical address;

Access time doubled

How to be faster?

cache!

Learn from History

- **Translation lookaside buffer (TLB)**

/translation buffer (TB)

a special **cache!**

that keeps (prev) address translations

- TLB entry

--tag: portions of the virtual address;

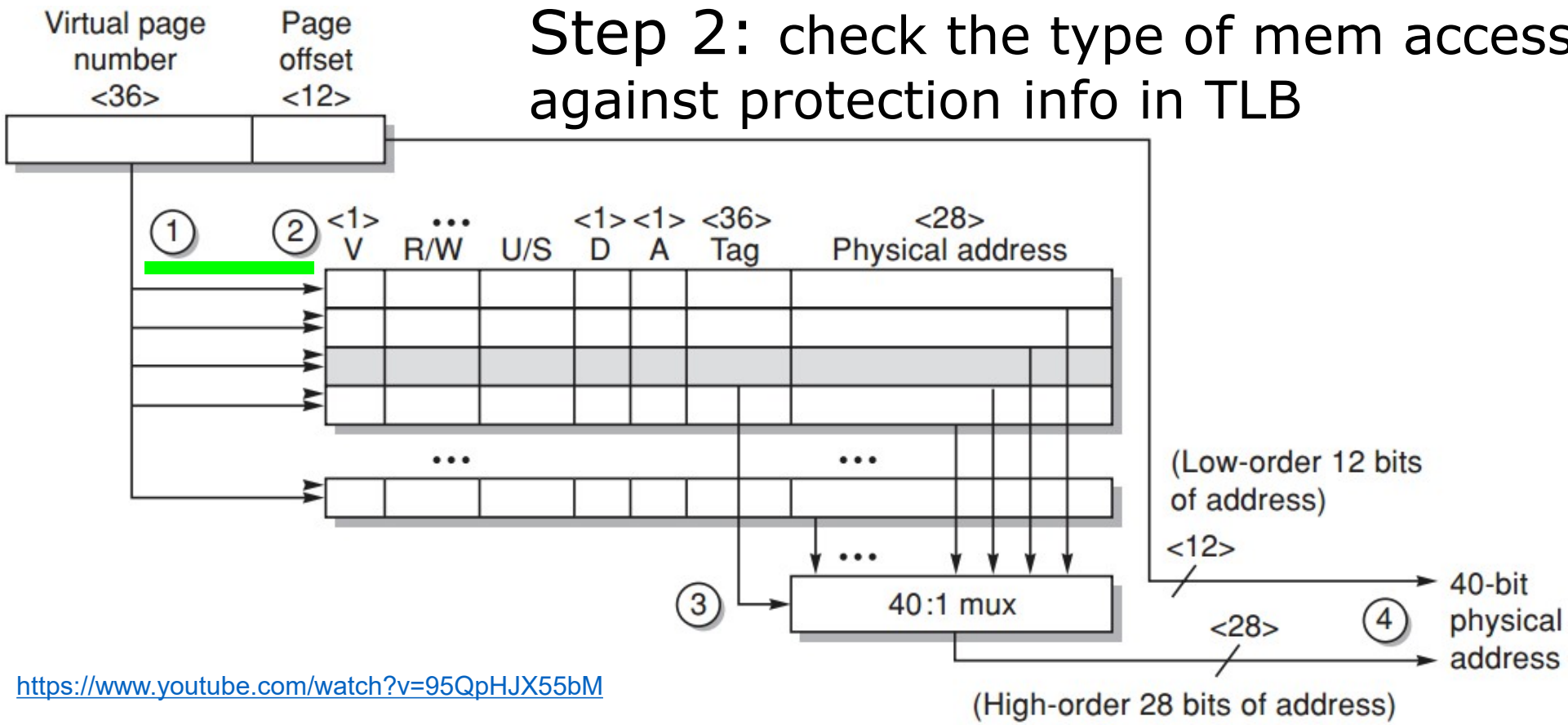
--data: a physical page frame number, protection field, valid bit, use bit, dirty bit;

TLB Example

• Opteron data TLB

Steps 1&2: send the virtual address to all tags

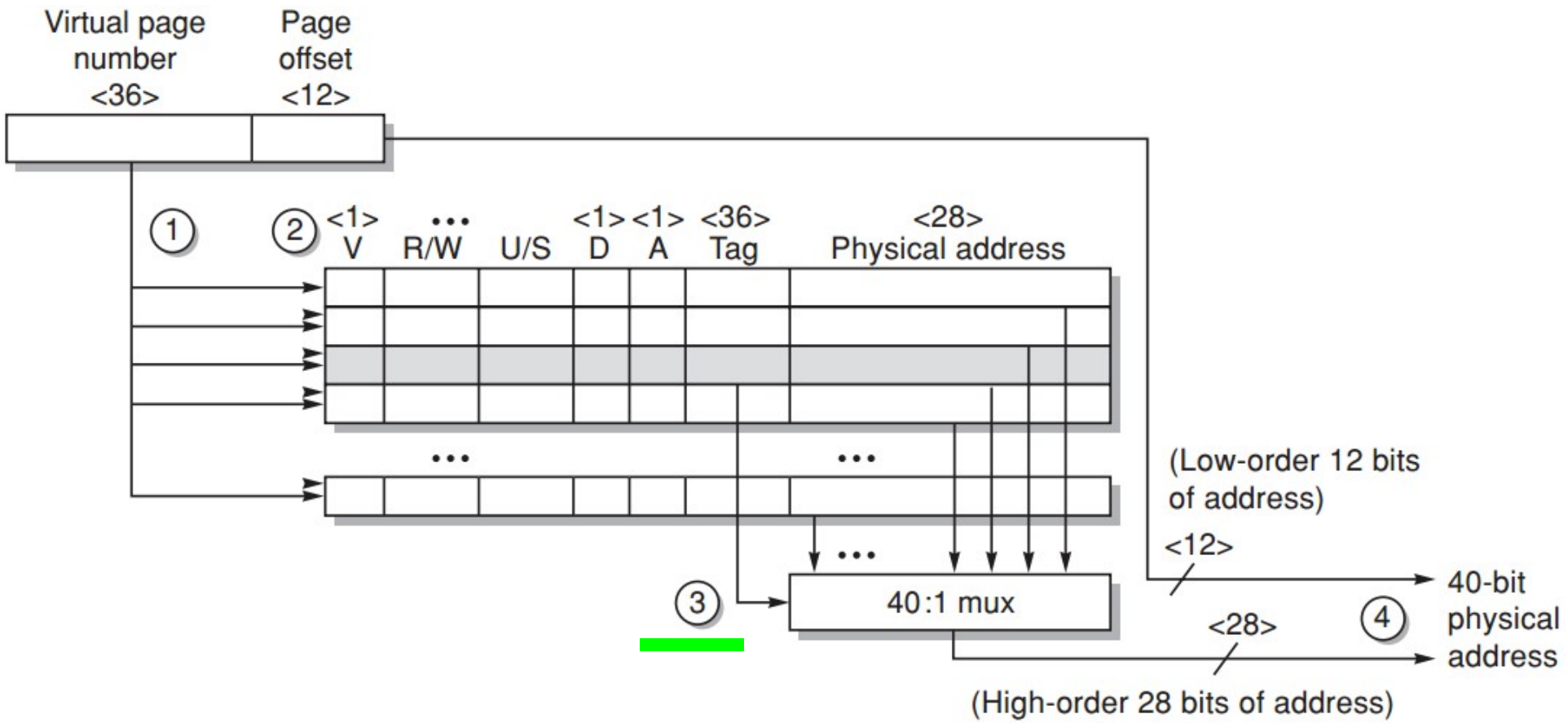
Step 2: check the type of mem access against protection info in TLB



TLB Example

• Opteron data TLB

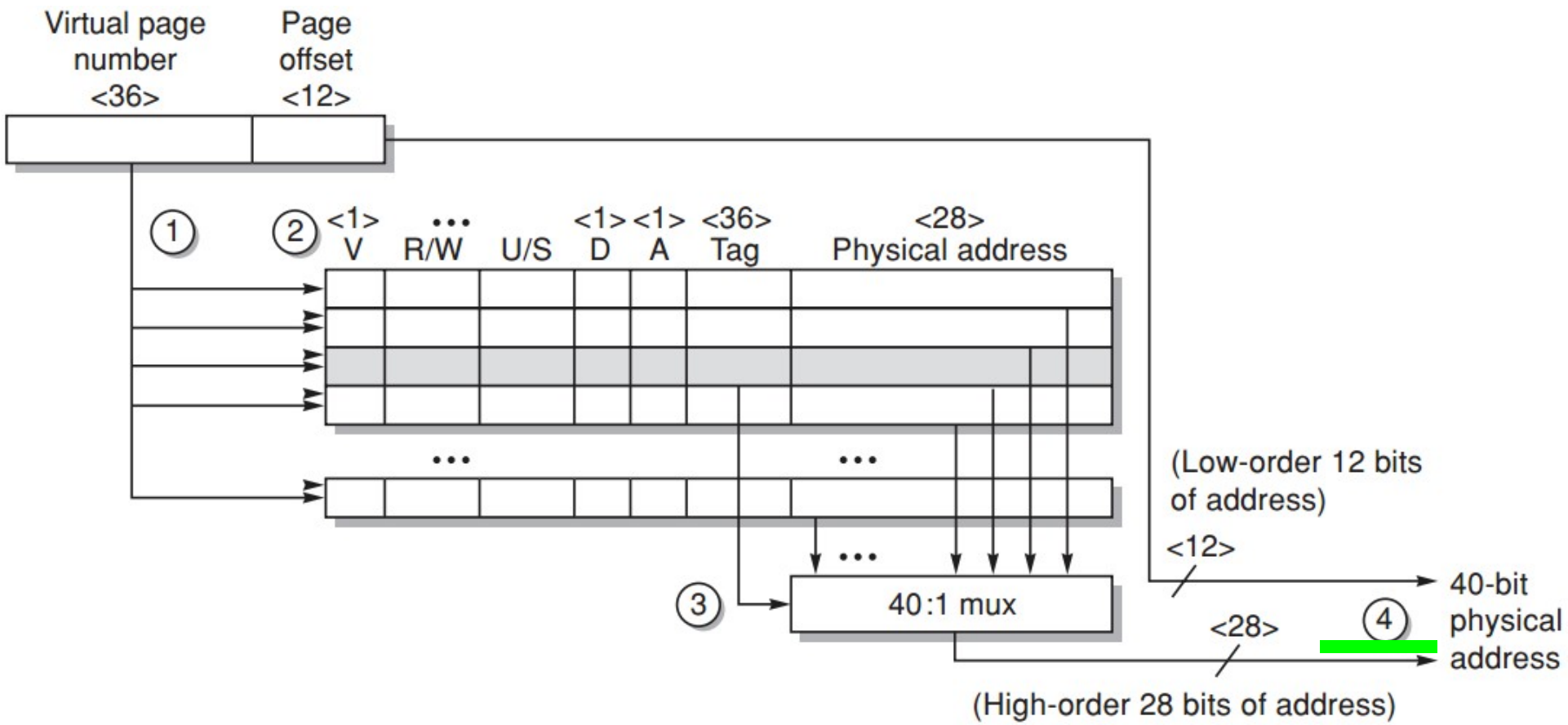
Steps 3: the matching tag sends phy addr through multiplexor



TLB Example

• Opteron data TLB

Steps 4: concatenate page offset to phy page frame to form final phy addr



Does page size matter?

Page Size Selection

Pros of **larger** page size

- Smaller page table, less memory (or other resources used for the memory map);
- Larger cache with fast cache hit;
- Transferring larger pages to or from secondary storage is more efficient than transferring smaller pages;
- Map more memory, reduce the number of TLB misses;

Page Size Selection

Pros of **smaller** page size

- Conserve storage

When a contiguous region of virtual memory is not equal in size to a multiple of the page size,

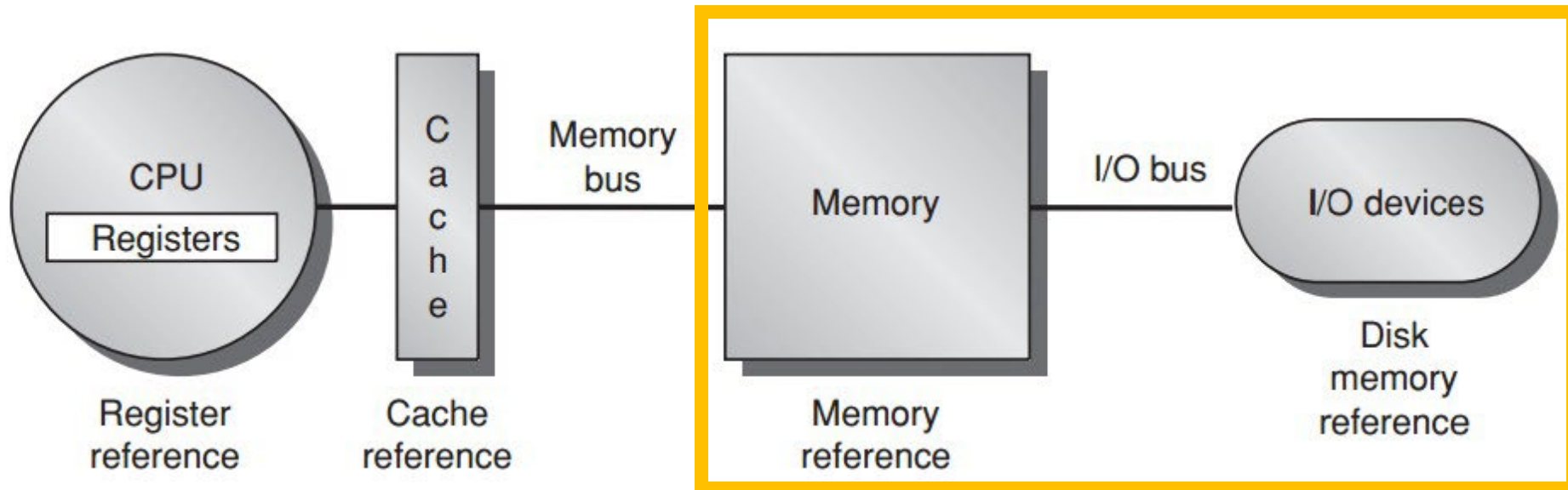
a small page size results in less wasted storage.

Page Size Selection

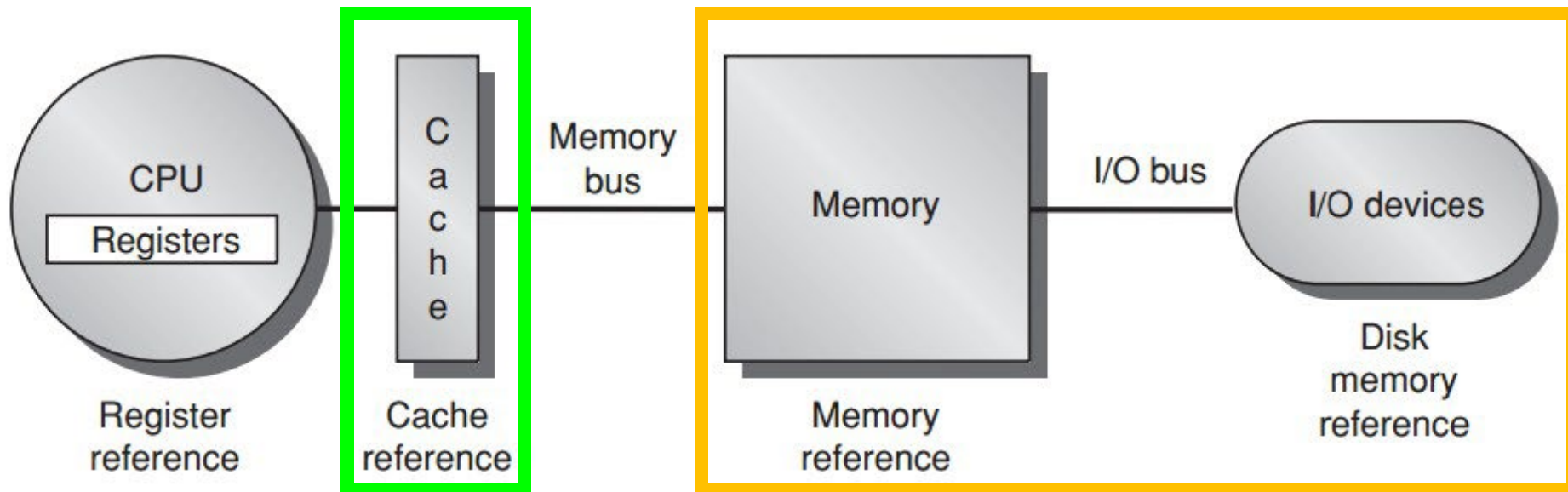
- Use both: **multiple page sizes**
- Recent microprocessors have decided to support multiple page sizes, mainly because of larger page size reduces the # of TLB entries and thus the # of TLB misses;

for some programs, TLB misses can be as significant on CPI as the cache misses;

Virtual Memory = Main Memory + Secondary Storage

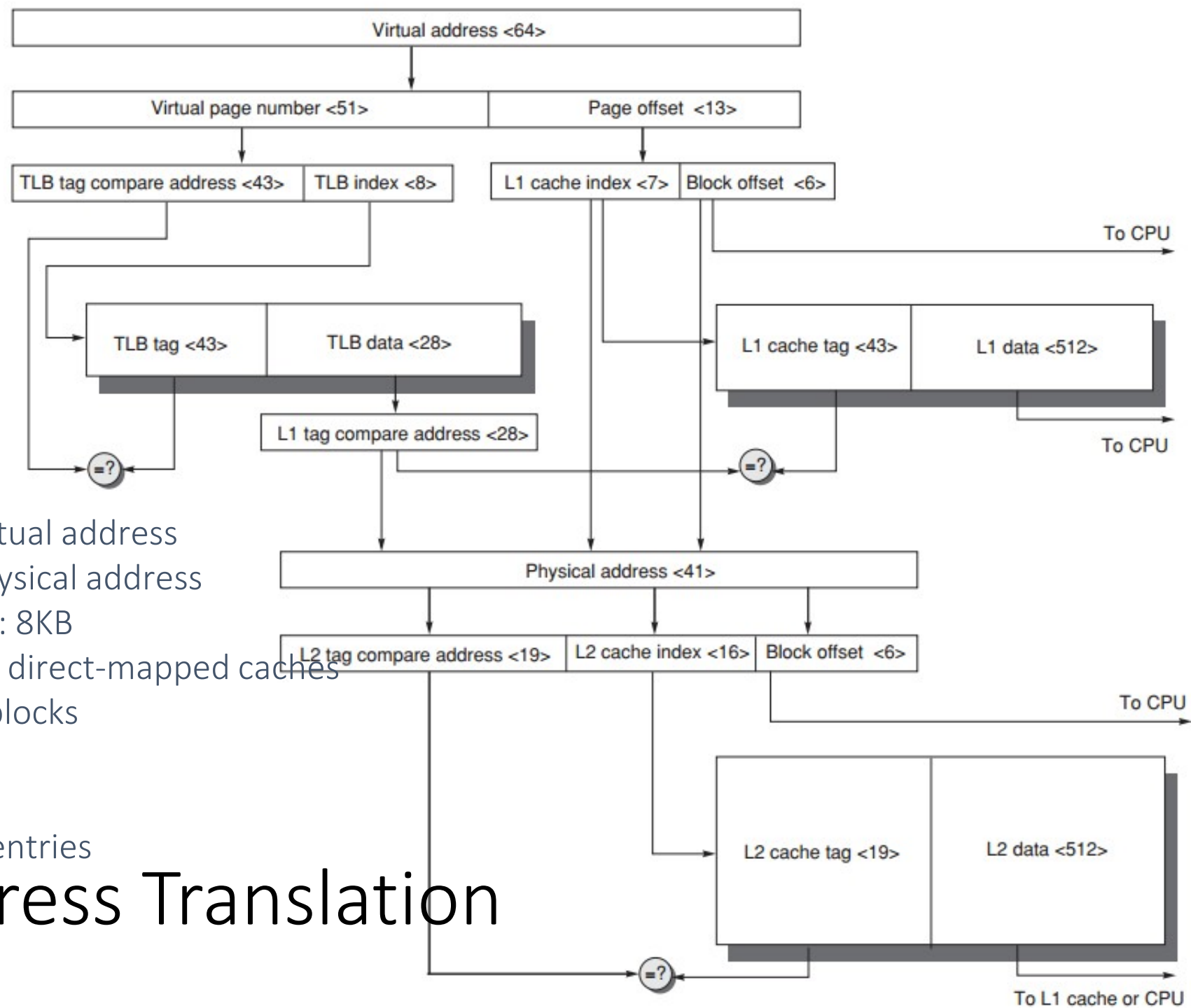


Virtual Memory = Main Memory + Secondary Storage



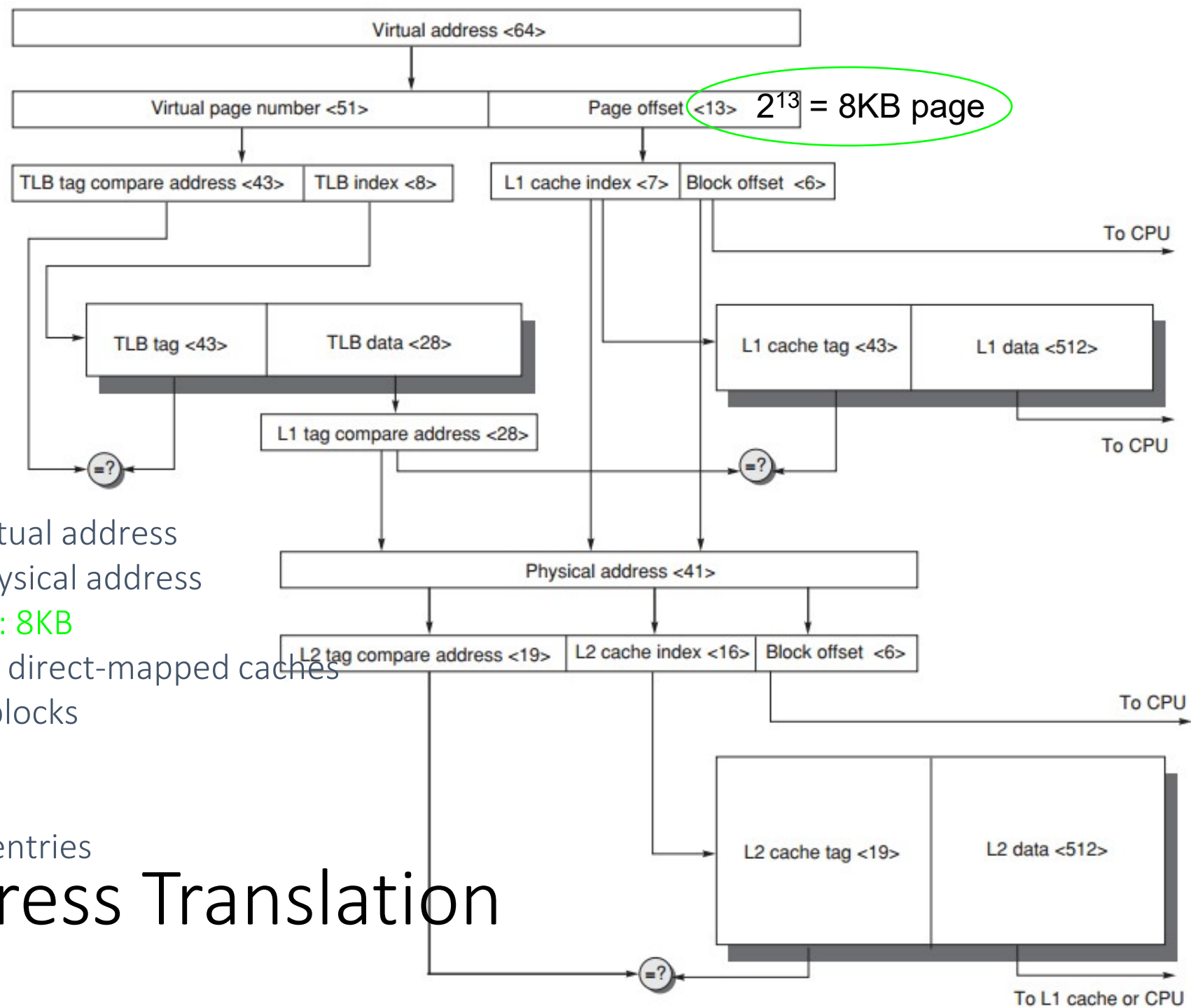
Address Translation: one more time, with cache

§ 3.5 Virtual Memory



- 64-bit virtual address
- 41-bit physical address
- page size: 8KB
- two-level direct-mapped caches
- 64-byte blocks
- L1: 8KB
- L2: 4MB
- TLB 256 entries

Address Translation



64-bit virtual address

41-bit physical address

page size: 8KB

two-level direct-mapped caches

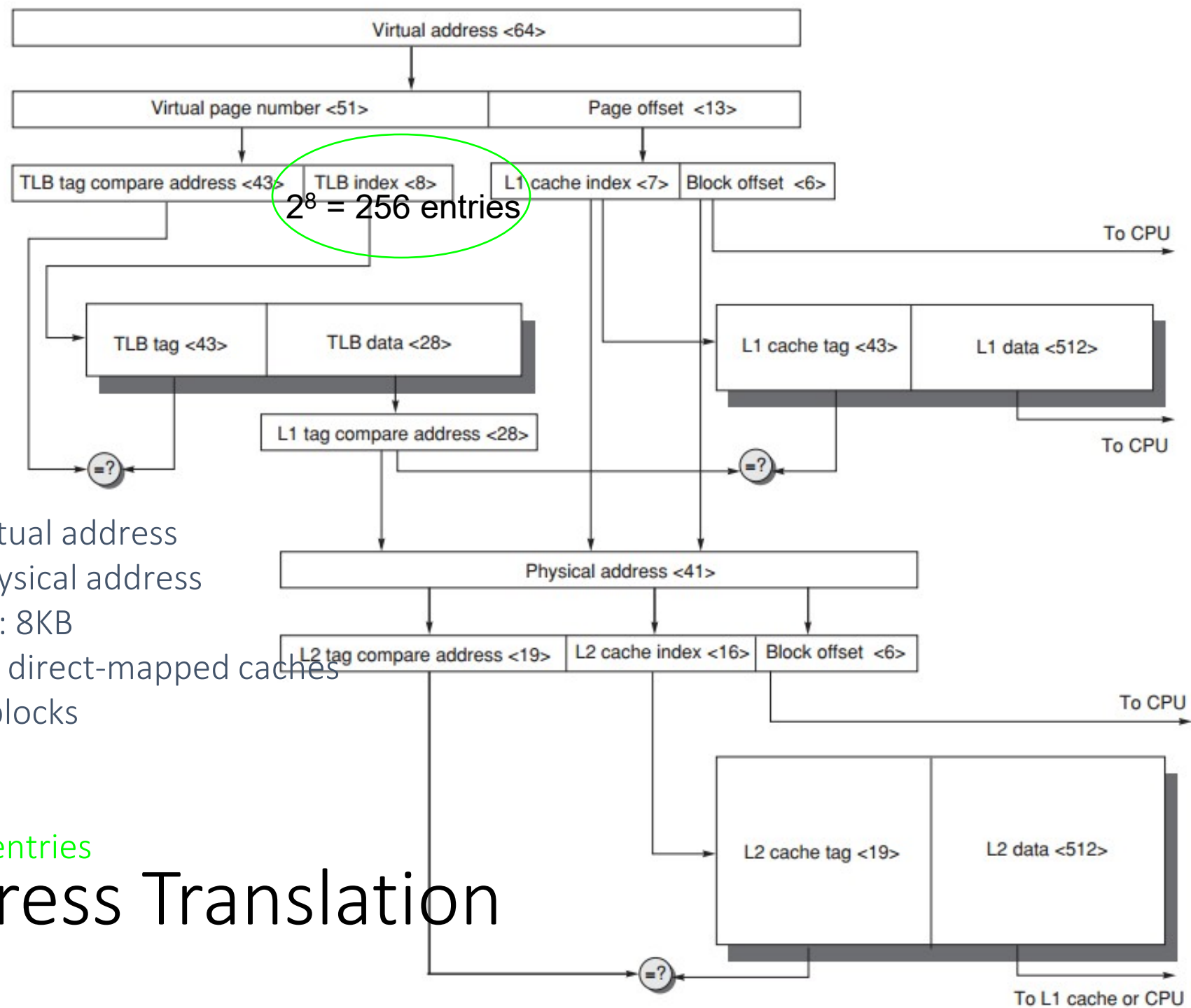
64-byte blocks

L1: 8KB

L2: 4MB

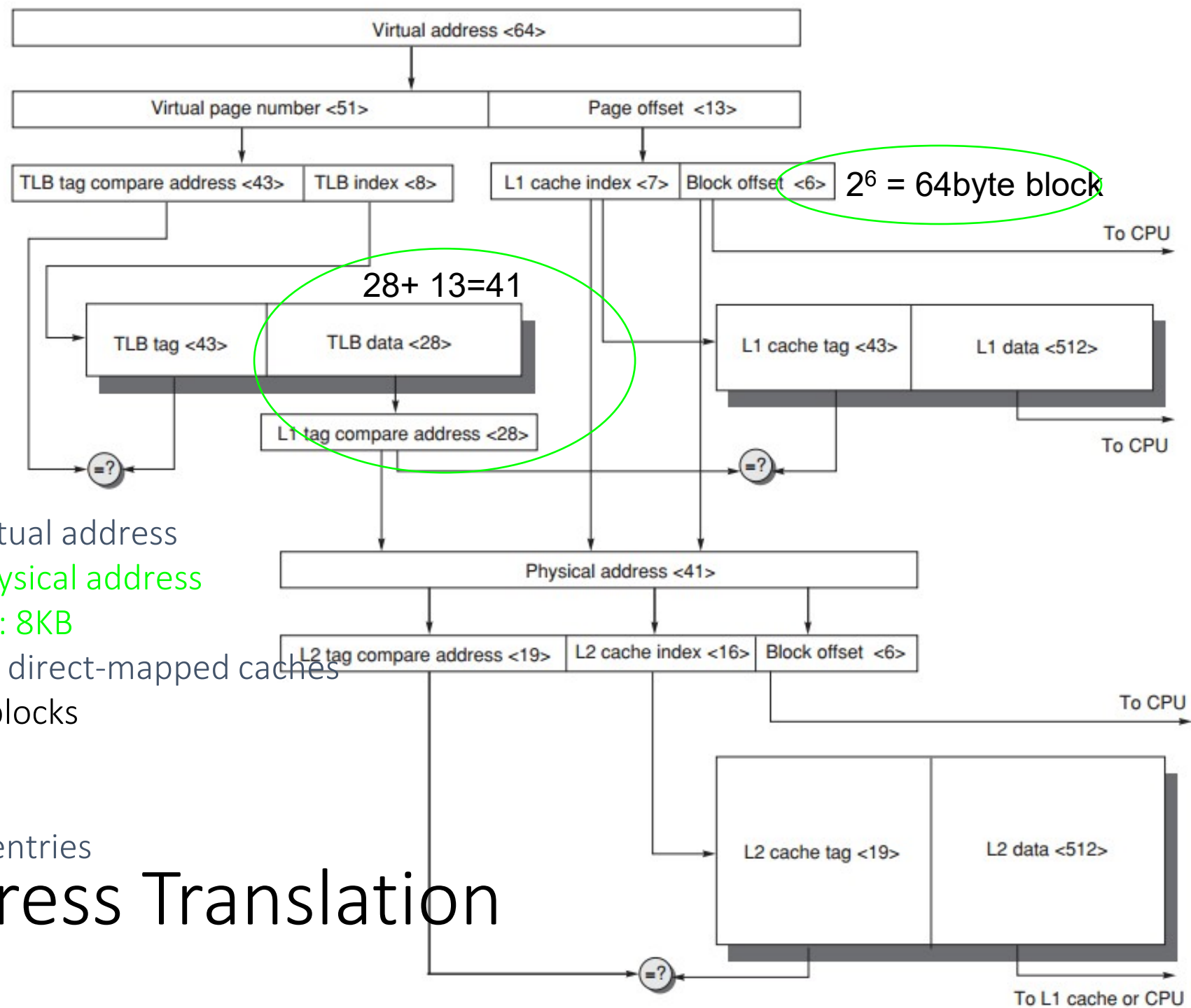
TLB 256 entries

Address Translation



64-bit virtual address
 41-bit physical address
 page size: 8KB
 two-level direct-mapped caches
 64-byte blocks
 L1: 8KB
 L2: 4MB
 TLB 256 entries

Address Translation



64-bit virtual address

41-bit physical address

page size: 8KB

two-level direct-mapped caches

64-byte blocks

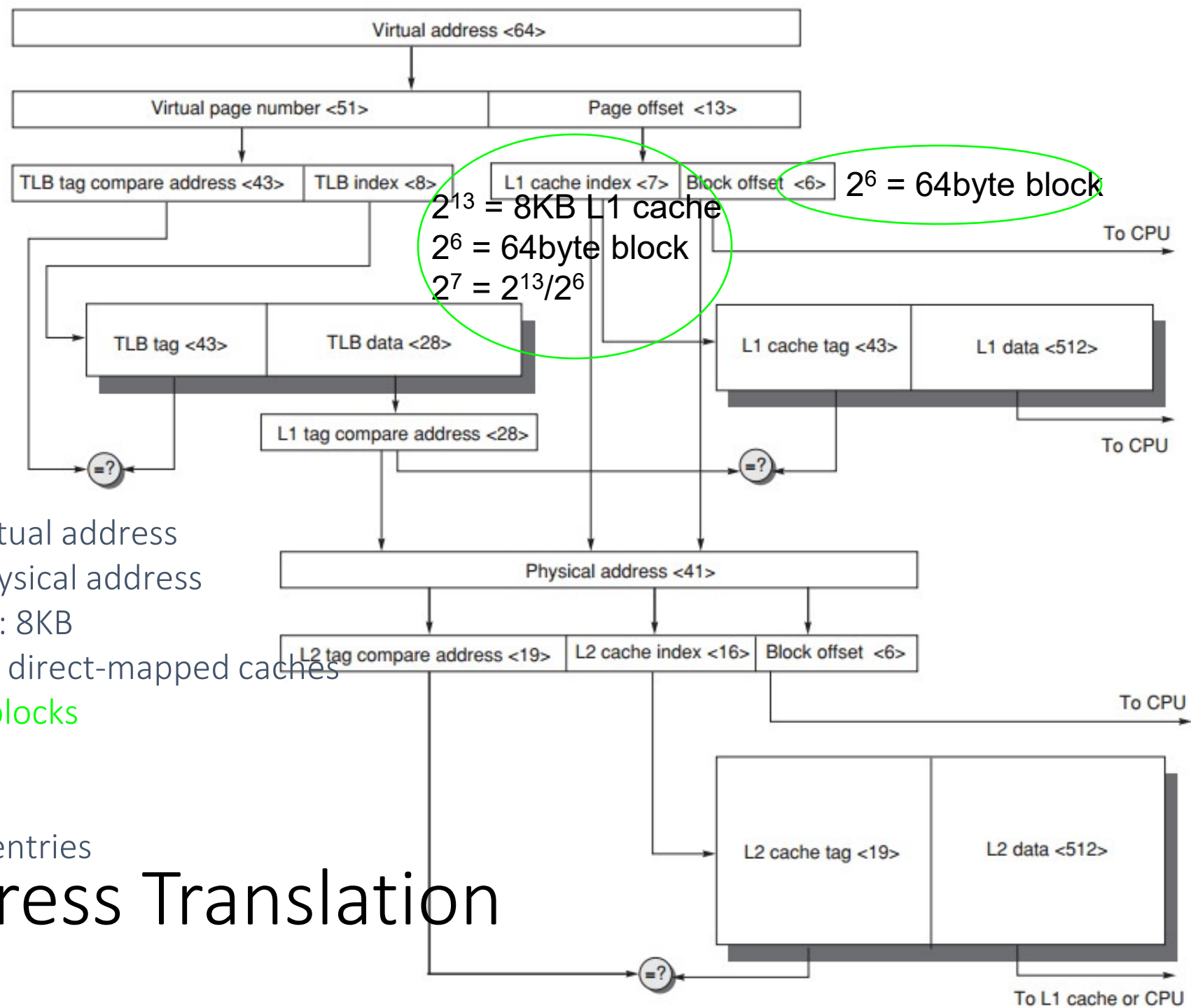
L1: 8KB

L2: 4MB

TLB 256 entries

Address Translation

§ 3.5 Virtual Memory

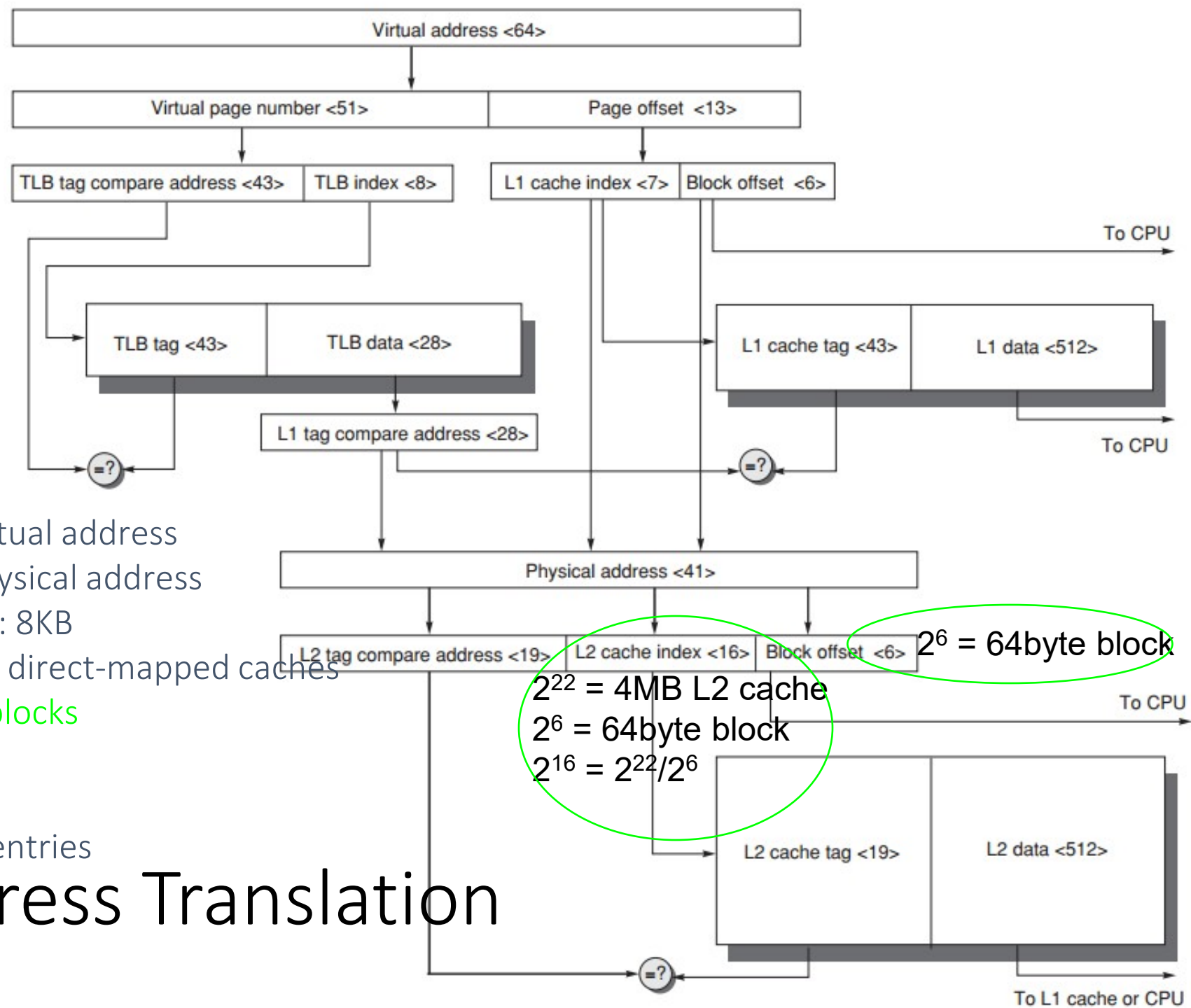


- 64-bit virtual address
- 41-bit physical address
- page size: 8KB
- two-level direct-mapped caches
 - L1: 8KB
 - L2: 4MB
- TLB 256 entries

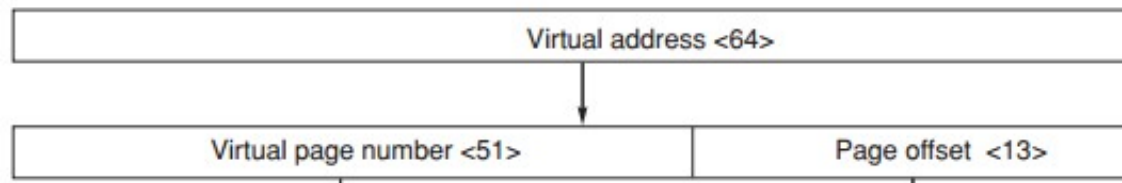
Address Translation

64-bit virtual address
 41-bit physical address
 page size: 8KB
 two-level direct-mapped caches
 64-byte blocks
 L1: 8KB
 L2: 4MB
 TLB 256 entries

Address Translation



ready?



64-bit virtual address

41-bit physical address

page size: 8KB

two-level direct-mapped caches

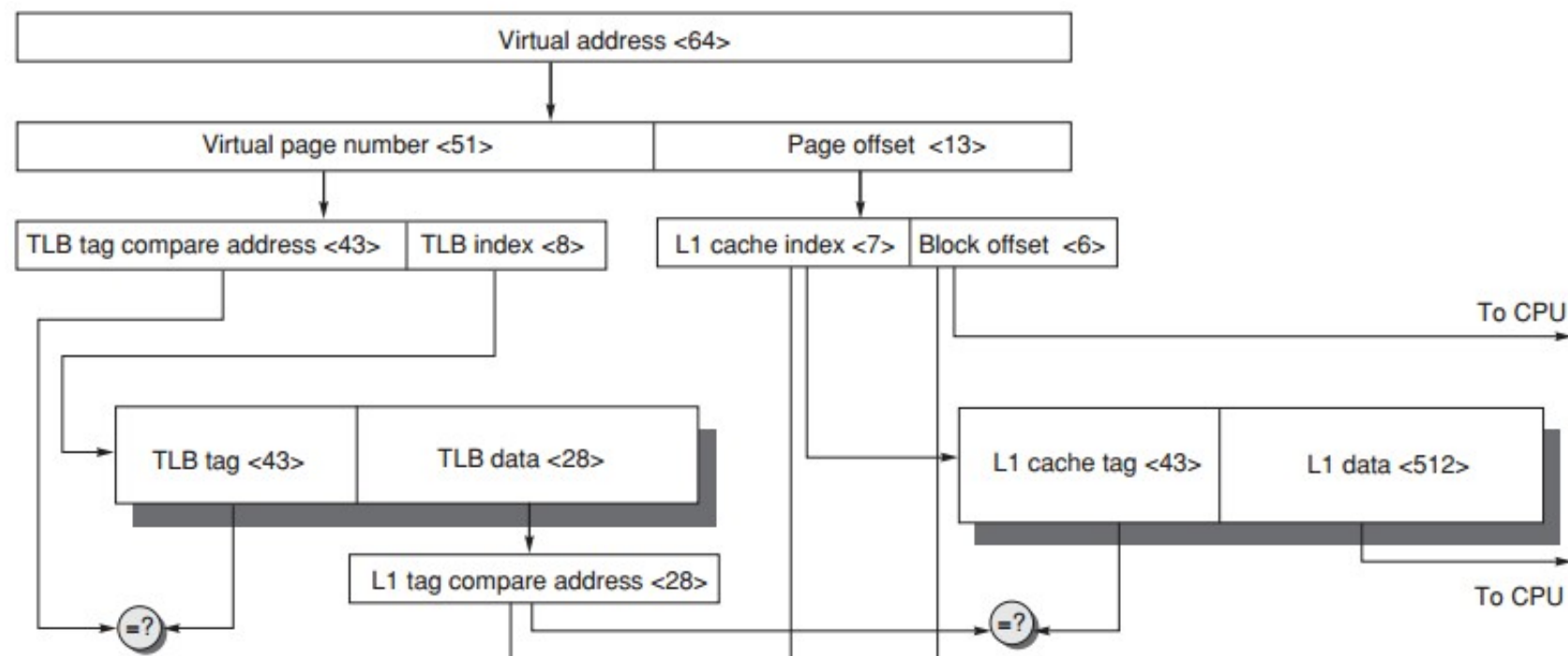
64-byte blocks

L1: 8KB

L2: 4MB

TLB 256 entries

Address Translation



64-bit virtual address

41-bit physical address

page size: 8KB

two-level direct-mapped caches

64-byte blocks

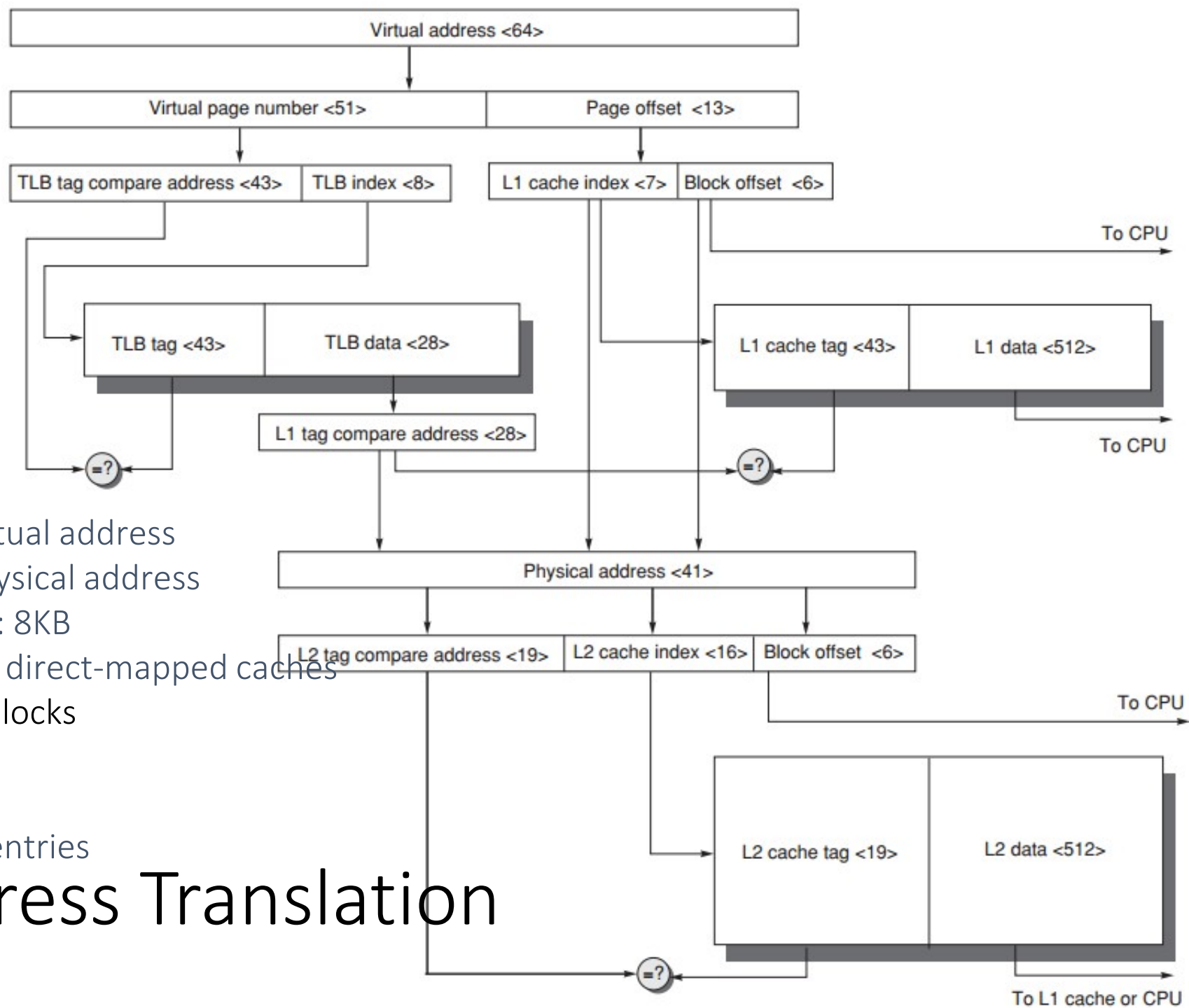
L1: 8KB

L2: 4MB

TLB 256 entries

Address Translation

§ 3.5 Virtual Memory



- 64-bit virtual address
- 41-bit physical address
- page size: 8KB
- two-level direct-mapped caches
- 64-byte blocks
- L1: 8KB
- L2: 4MB
- TLB 256 entries

Address Translation

**You said virtual memory
promised safety?**

**You said virtual memory
promised safety?**
mem protection & sharing among programs

Multiprogramming

- Enable a computer to be shared by several programs running concurrently
- Need protection and sharing among programs

Process

- A running program plus any state needed to continue running it
- **Time-sharing**
shares processor and memory with interactive users simultaneously;
gives the illusion that all users have their own computers;
- **Process/context switch**
from one process to another

Process

- Maintain **correct process behavior**
 - computer designer must ensure that the processor portion of the process state can be saved and restored;
 - OS designer must guarantee that processes do not interfere with each others' computations;
- Partition main memory so that several different processes have their state in memory at the same time

Process Protection

- **Proprietary page tables**

processes can be protected from one another by having their own page tables,

each pointing to distinct pages of memory;

user programs must be prevented from modifying their page tables

Process Protection

- **Rings**

added to the processor protection structure,
expands memory access protection to multiple levels.

- The most trusted accesses anything
- The second most trusted accesses everything except the innermost level
- ...
- The civilian programs are the least trusted, have the most limited range of accesses.

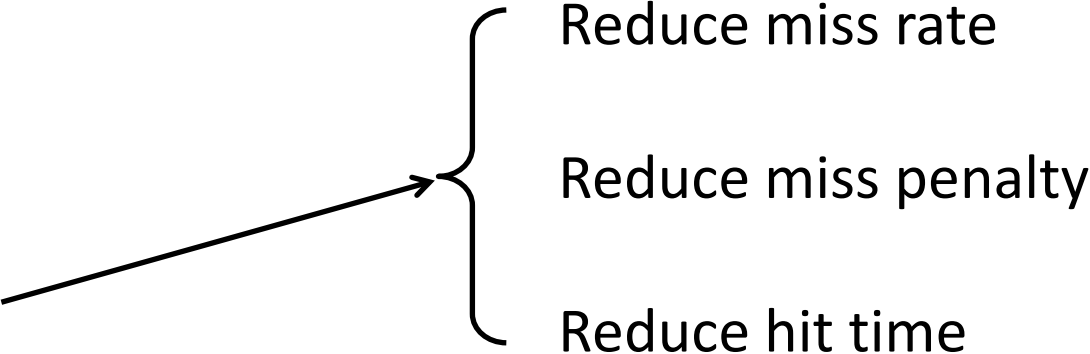
Process Protection

- **Keys and Locks**

a program cannot unlock access to the data unless it has the key

- For keys/capabilities to be useful, hardware and OS must be able to explicitly pass them from one program to another without allowing a program itself to forge them

Summary

- Memory hierarchy
 - From single level to multi level
 - Evaluate the performance parameters of the storage system (average price per bit C ; hit rate H ; average memory access time T)
 - Cache basic knowledge
 - Mapping rules
 - Access method
 - Replacement algorithm
 - Write strategy
 - Cache performance analysis
- 
- Virtual Memory (the influence of memory organization structure on Cache failure rate)



knowledge map

