Chapter 3

Memory Hierarchy

- **Technology Trend and Memory Hierarchy**

- **Four Questions for Cache Designers**

- **Memory System Performance**
  - **Improve Cache Performance**

- **Virtual Memory**

Memory

- Register

- Cache

- Memory

- Storage

- Mechanical memory
- Electronic memory
  - SRAM
  - DRAM
    - SDRAM
    - DDR
  - GDRAM
    - GDDR
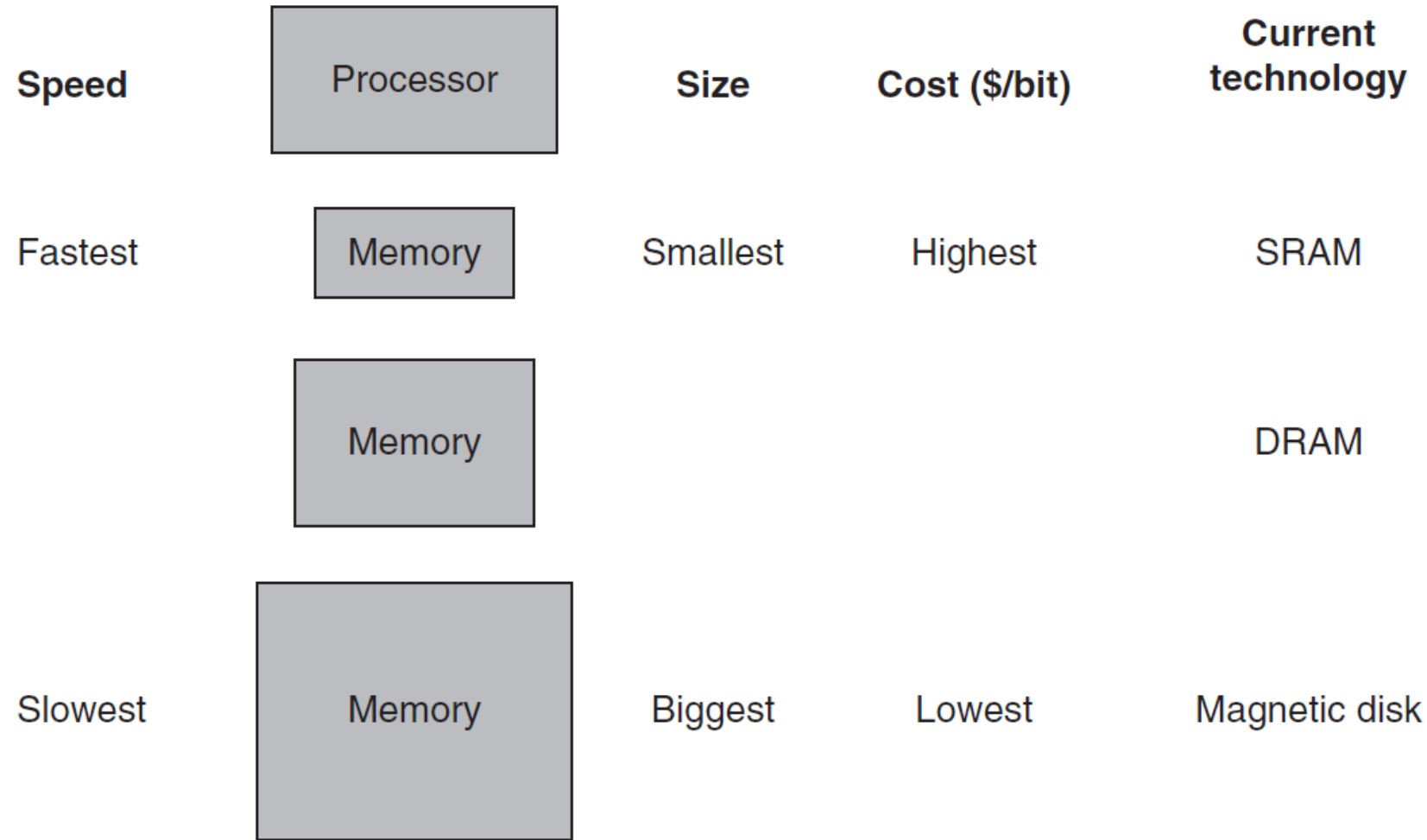 - HBM
  - EEPROM
    - NAND
    - NOR
- Optical memory

Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.

Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.
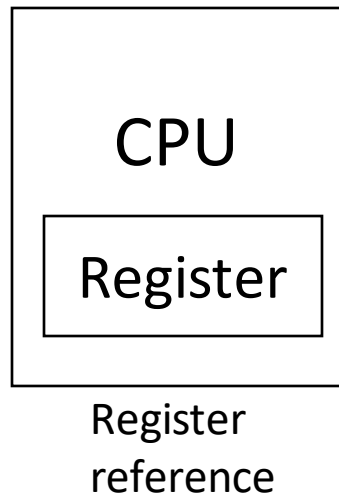
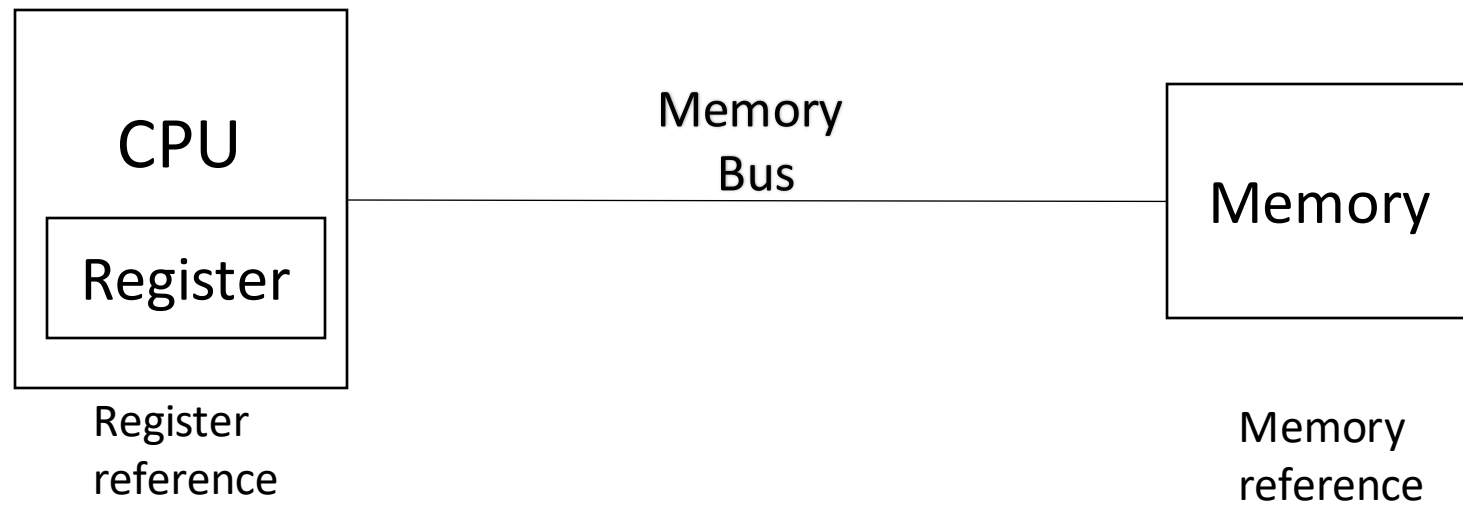# Memory?

- Load   R2, 0(R1)

- Store   R2, 0(R1)

Speed		Size	Cost ($/bit)	Current technology
	Processor			
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

Data processing & temporary storage

CPU

Register

Register
reference

Temporary storage

CPU

Register

Register
reference

Memory
Bus

Memory

Memory
reference

Faster temporary storage

CPU

Register

Memory
Bus

Memory

Storage

Register
reference

Memory
reference

Flash
memory
reference

Size
Speed

Faster temporary storage

CPU		Cache		Memory Bus		Memory		Storage

CPU

Register

Cache

Memory
Bus

Memory

Storage

Register
reference

Cache
reference

Memory
reference

Flash
memory
reference

Size
Speed

Memory Hierarchy

CPU	L1 Cache	L2 Cache	Memory Bus	Memory	Storage
Register					

Register reference	Level 1 Cache reference	Level 2 Cache reference	Memory reference	Flash memory reference

Size：	1000 bytes	64KB	256KB	1-2GB	4-64GB
Speed:	300ps	1ns	5-10ns	50-100ns	25-50us

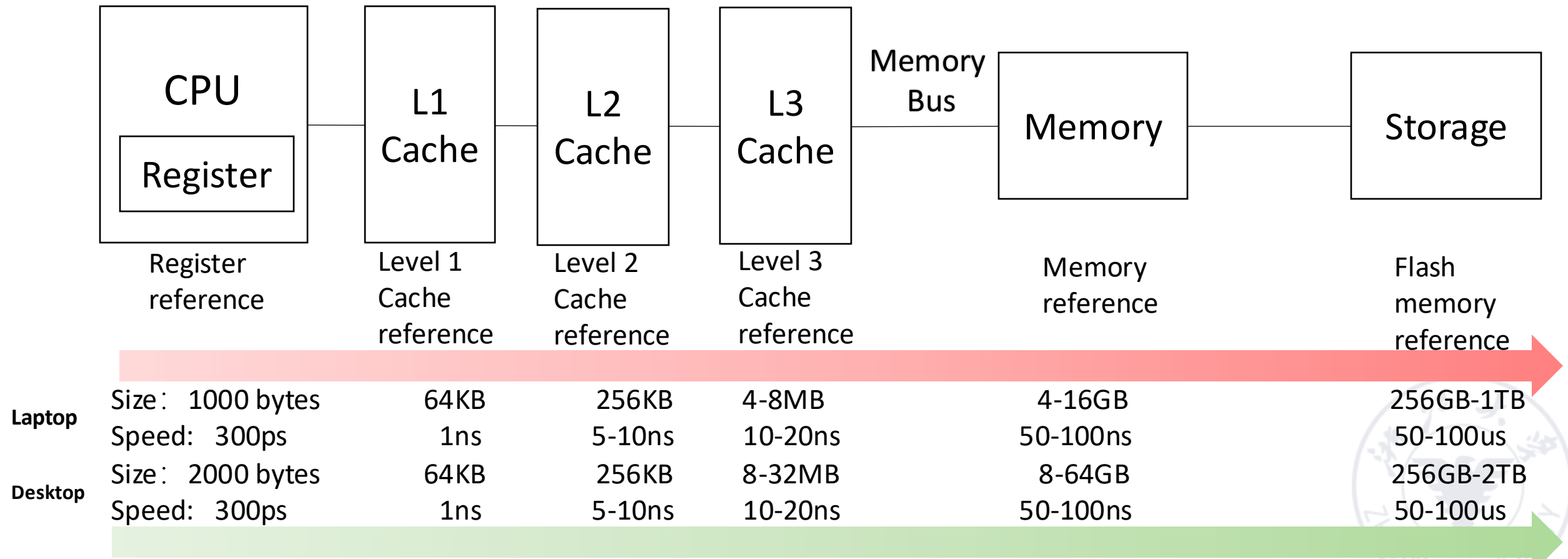**Memory hierarchy for a personal mobile device**

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

Memory Hierarchy

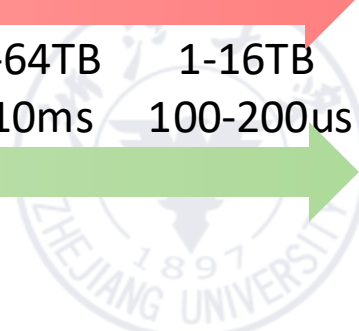| CPU Register	L1 Cache	L2 Cache	L3 Cache	Memory Bus	Memory	Storage
Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference		Memory reference	Flash memory reference

Laptop	Size： 1000 bytes Speed: 300ps	64KB 1ns	256KB 5-10ns	4-8MB 10-20ns		4-16GB 50-100ns	256GB-1TB 50-100us
Desktop	Size： 2000 bytes Speed: 300ps	64KB 1ns	256KB 5-10ns	8-32MB 10-20ns		8-64GB 50-100ns	256GB-2TB 50-100us

Memory hierarchy for a laptop or a desktop

Memory Hierarchy

CPU	L1 Cache	L2 Cache	L3 Cache	Memory Bus	Memory	I/O Bus	Disk storage
Register							Flash storage

Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference	Memory reference	Disk memory reference	Flash memory reference

Size： 4000 bytes   64KB   256KB   16-64MB   32-256GB   16-64TB   1-16TB

Speed: 200ps   1ns   3-10ns   10-20ns   50-100ns   5-10ms   100-200us
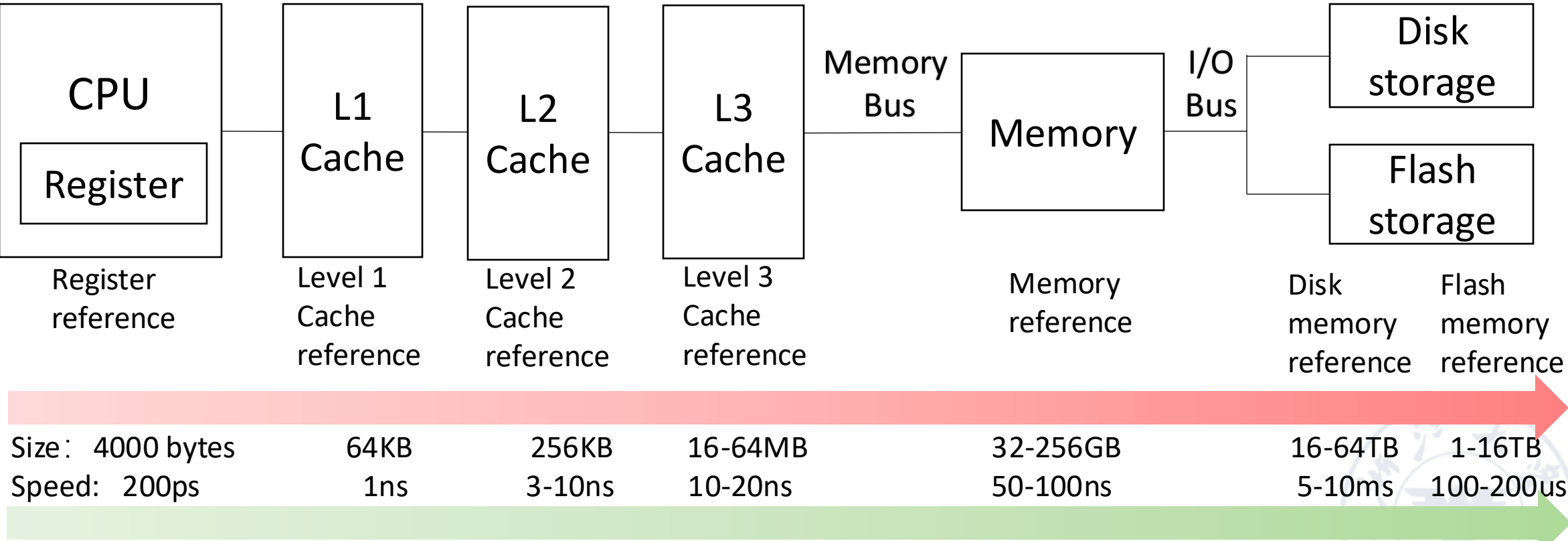
Memory hierarchy for server

Wait, but what's cache?

data request?

Wait, but what's cache?

program/ instructions

in / out?

data request?

Wait, but what's cache?

program/ instructions

in / out?

data request?

program/ instructions

Wait, but what's cache?

optimization?   in / out?
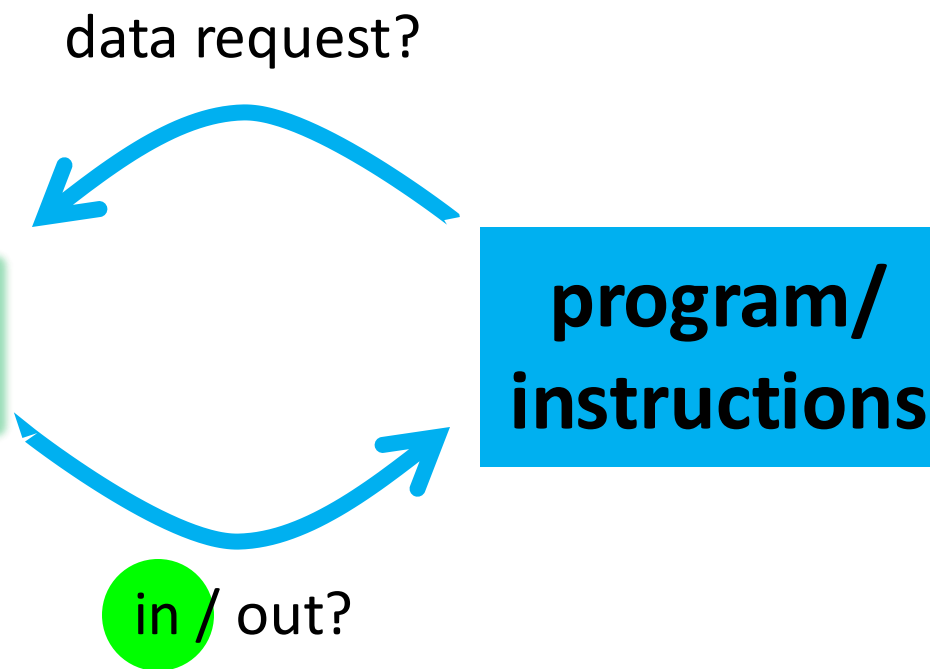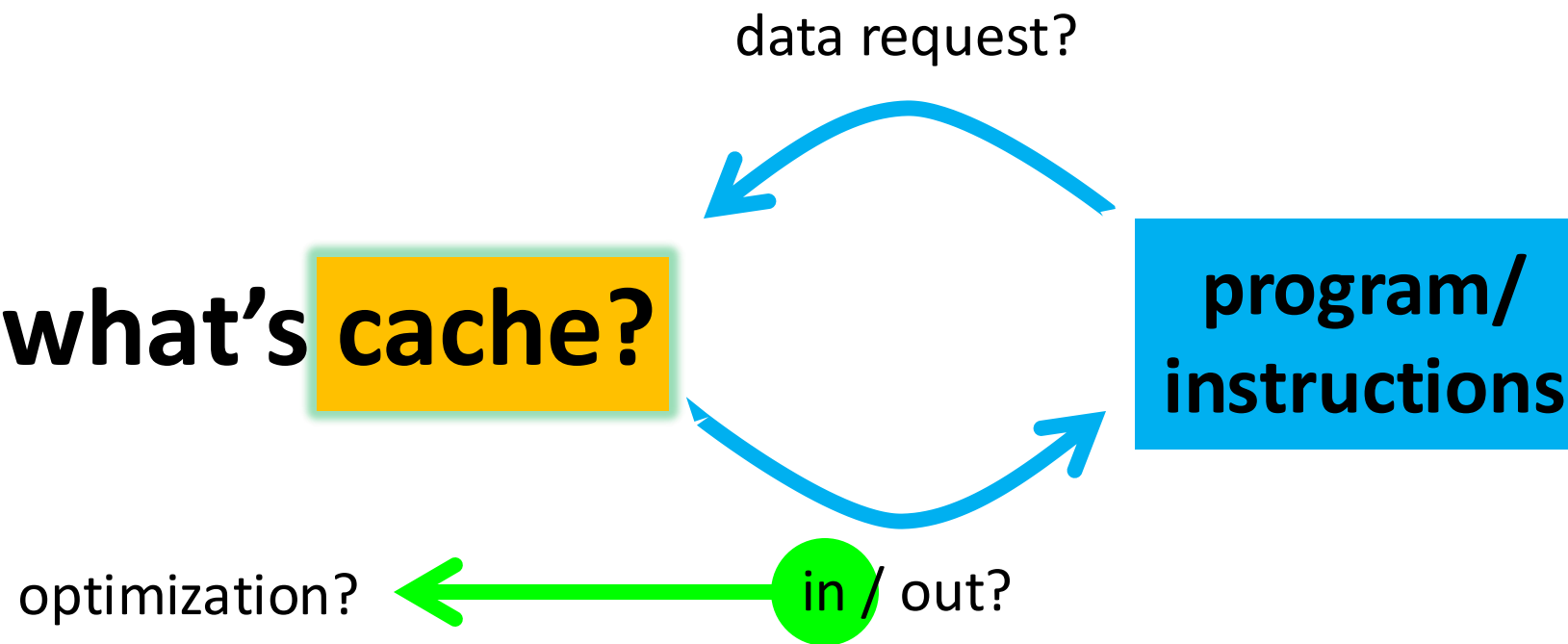
So, what's cache?

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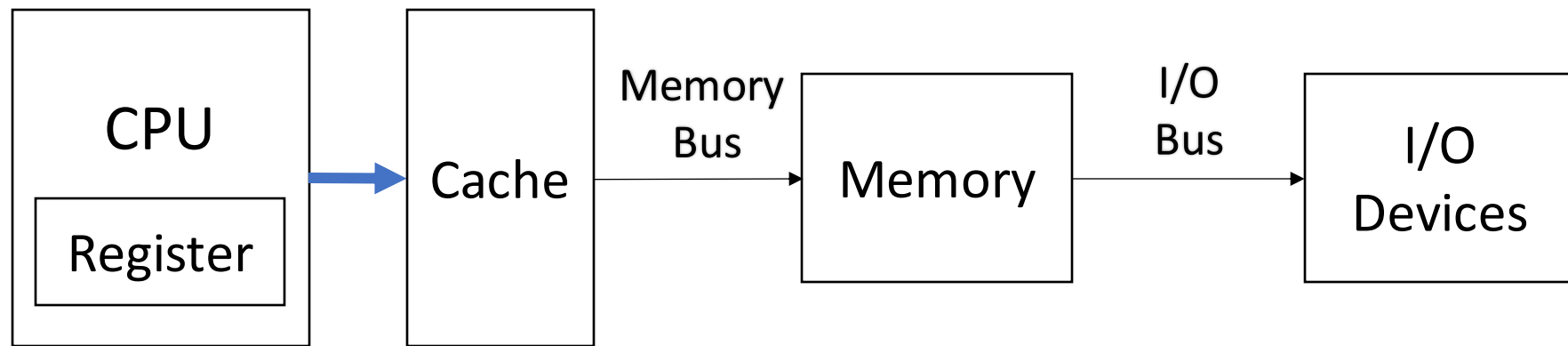
- The highest or first level of the memory hierarchy encountered once the *addr* leaves the processor
- Employ buffering to reuse commonly occurring items

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

cache

requested word

memory

Block/Line Run

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

cache

requested word

requested word

memory

Block/Line Run

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

cache

requested word

in a block

requested word

in a block
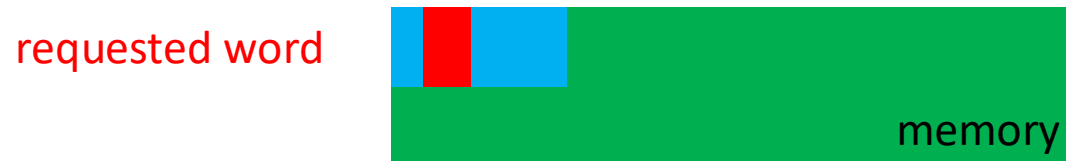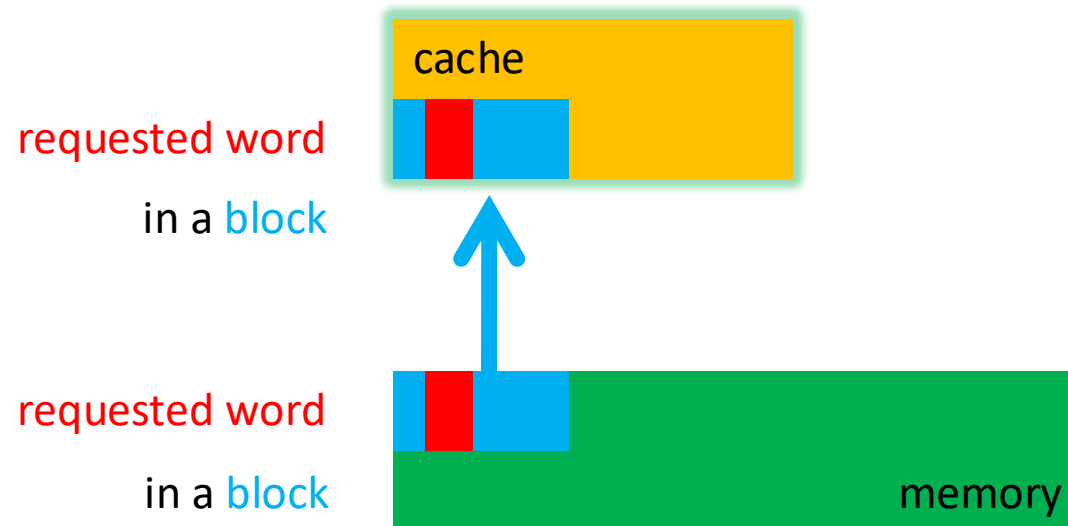
memory
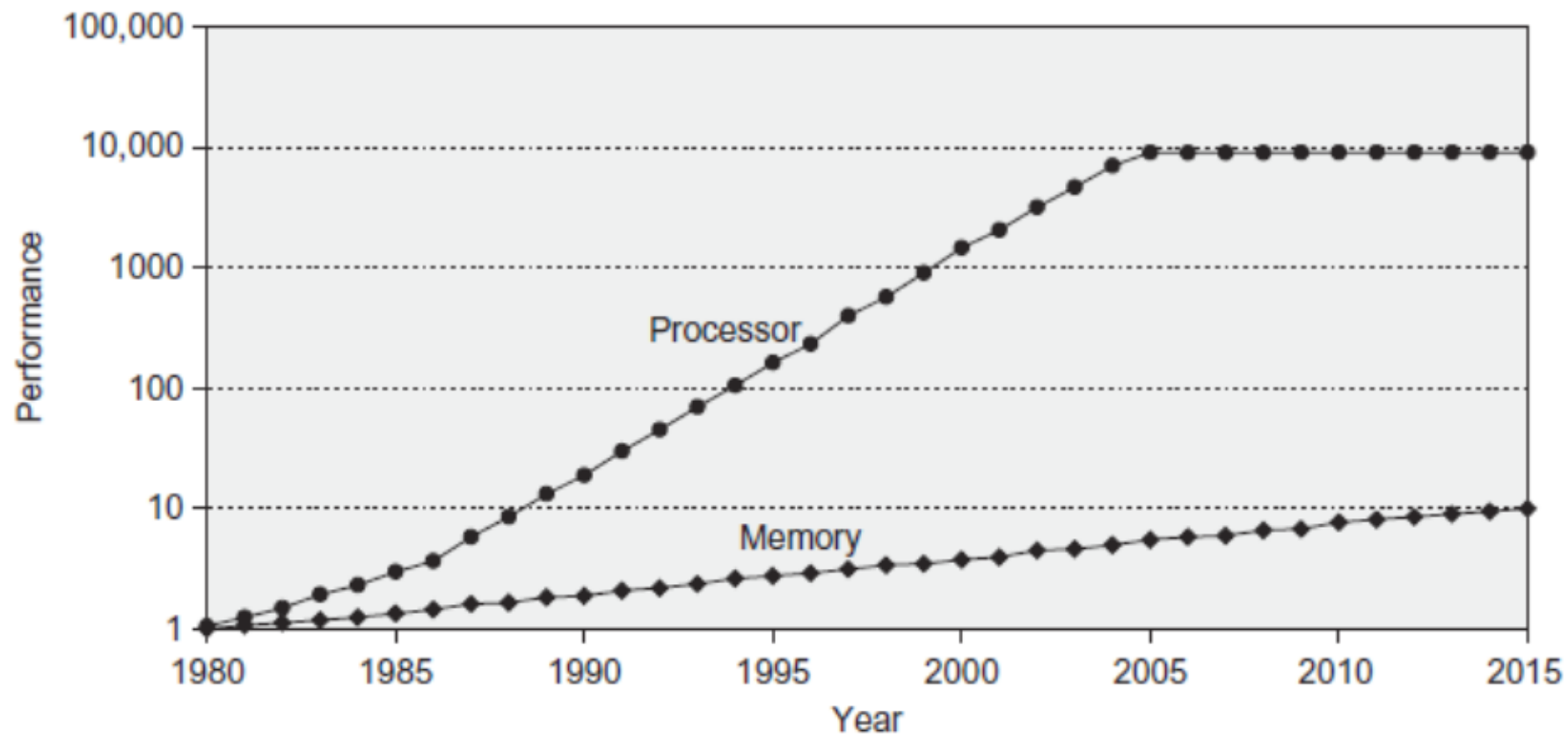
Technology Trend

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

SRAM (*static random access memory*)

DRAM (*dynamic random access memory*)
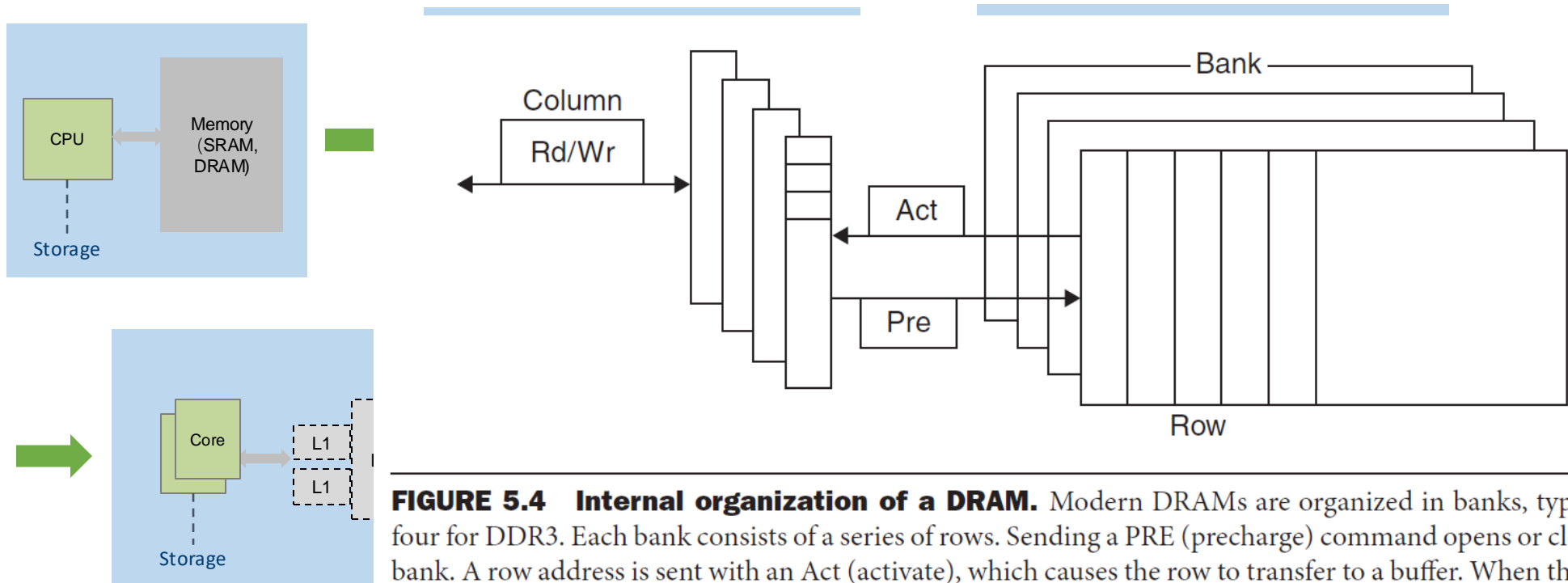
SDRAM (Synchronous DRAM)

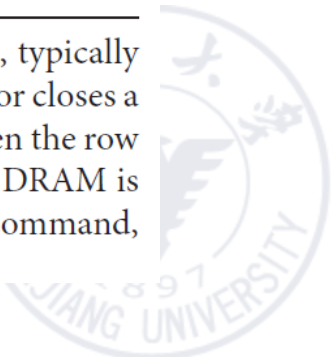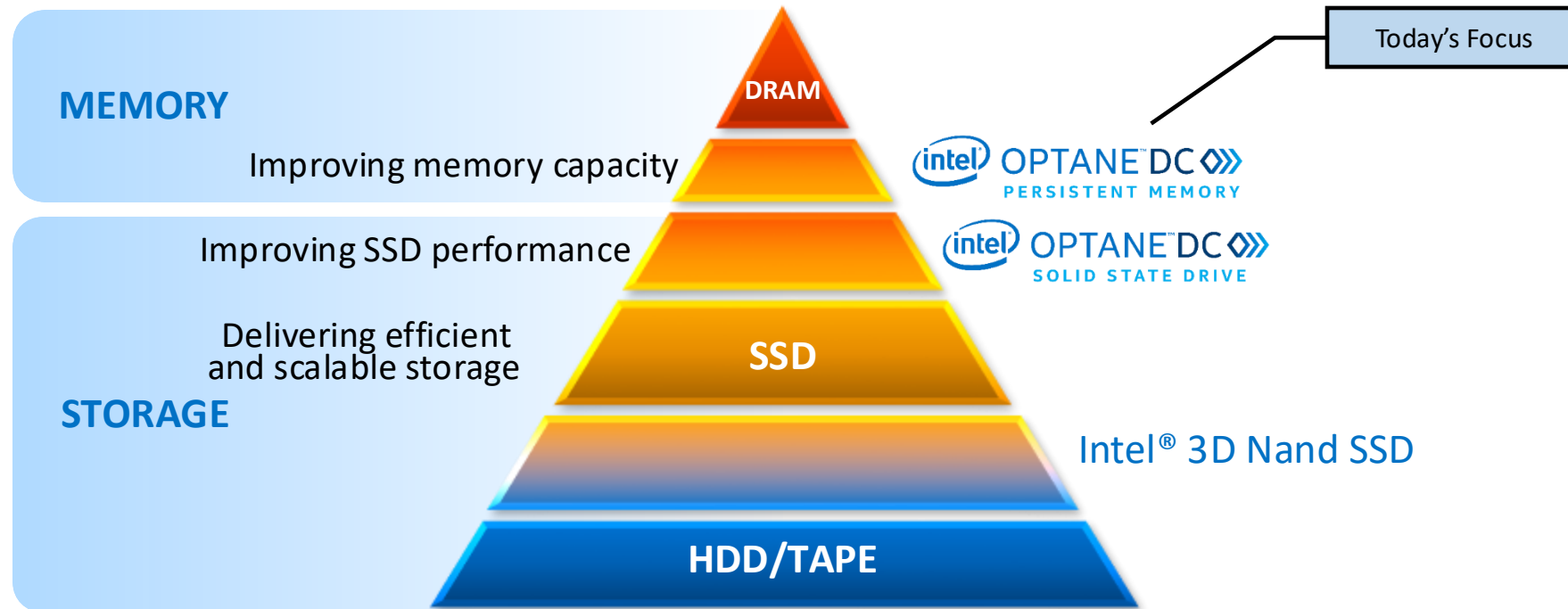*Double Data Rate (DDR)* SDRAM

FIGURE 5.4   Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.
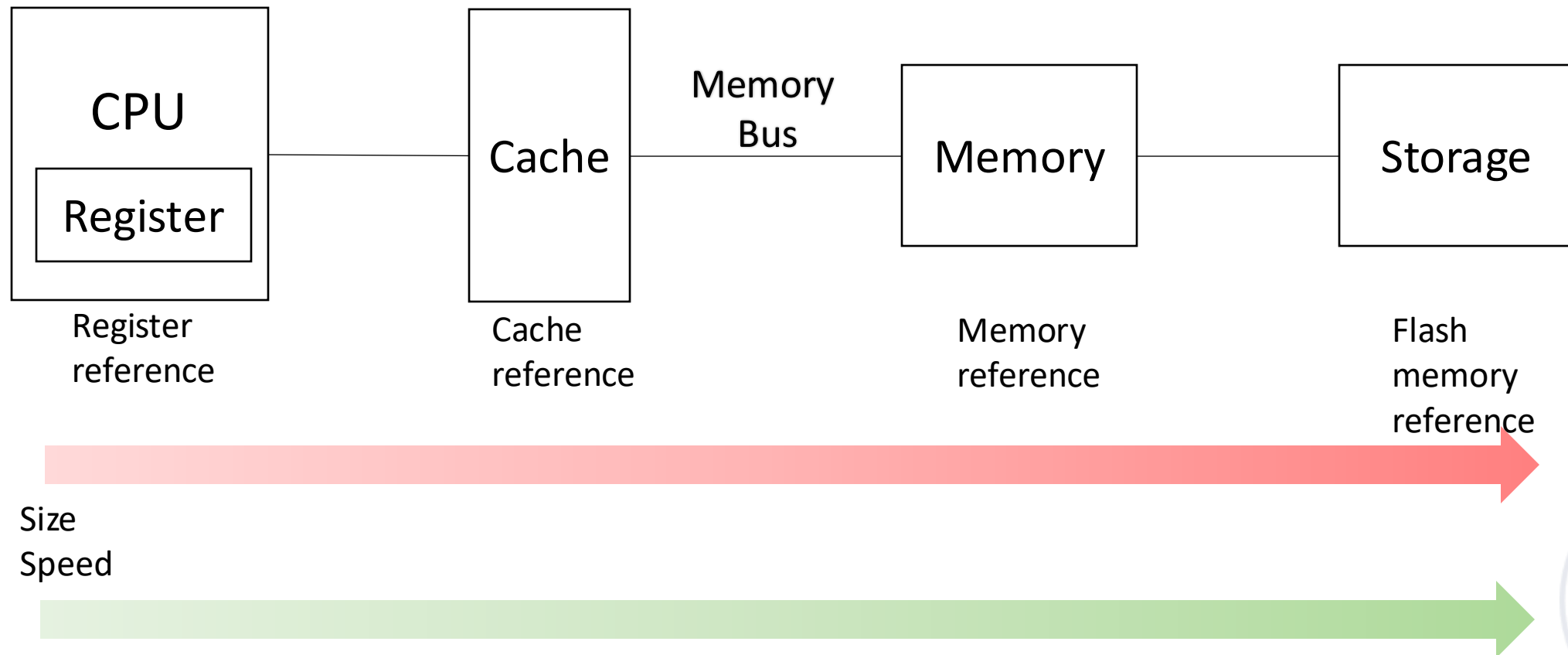
HBM (*High Bandwidth Memory*)

Memory/Storage

Today's Focus

MEMORY

DRAM

Improving memory capacity

(intel) OPTANE™DC ◊》
PERSISTENT MEMORY

Improving SSD performance

(intel) OPTANE™DC ◊》
SOLID STATE DRIVE

Delivering efficient and scalable storage

**SSD**

STORAGE

Intel® 3D Nand SSD

HDD/TAPE

Processor-Memory Performance Gap

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.

Speed (ns):	1s	10s	100s	10,000,000s (10s ms)
Size (bytes):	100s	Ks	Ms	Gs
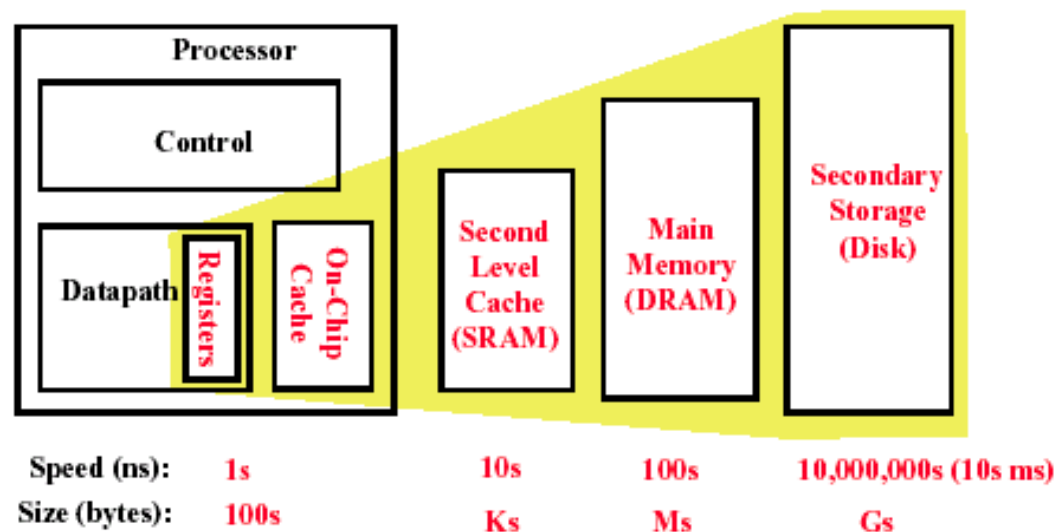
Cache Locality

- **Temporal locality**

  need the requested word again soon

- **Spatial locality**

  likely need other data within the same 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

- Causes of Cache misses

  ➢**Compulsory:** first reference to a block

  ➢**Capacity**: blocks discarded and later retrieved

  ➢**Conflict:** program makes repeated references to multiple addresses from different blocks that map to the same location in the cache

Enhance performance

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.

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.
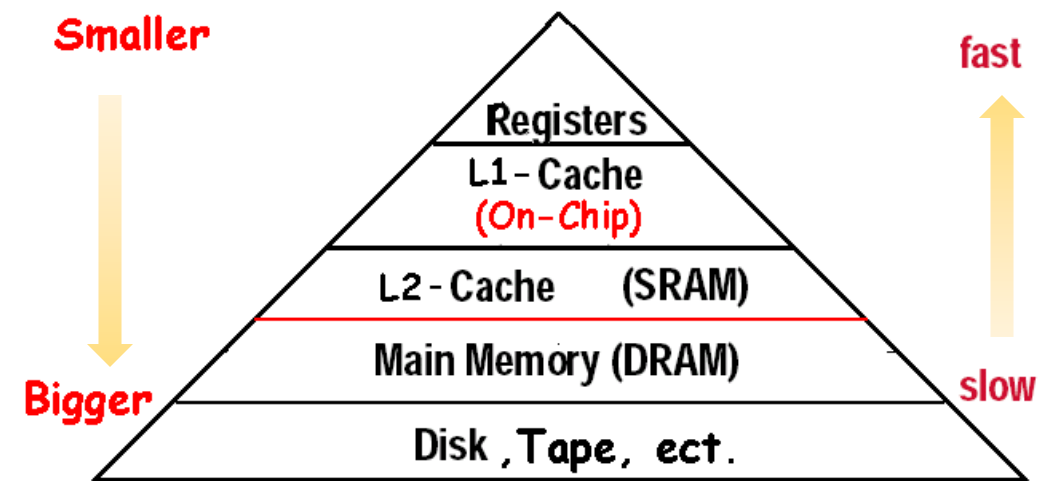 - Main memory is very small
    - often no disk storage

36 terms of Cache

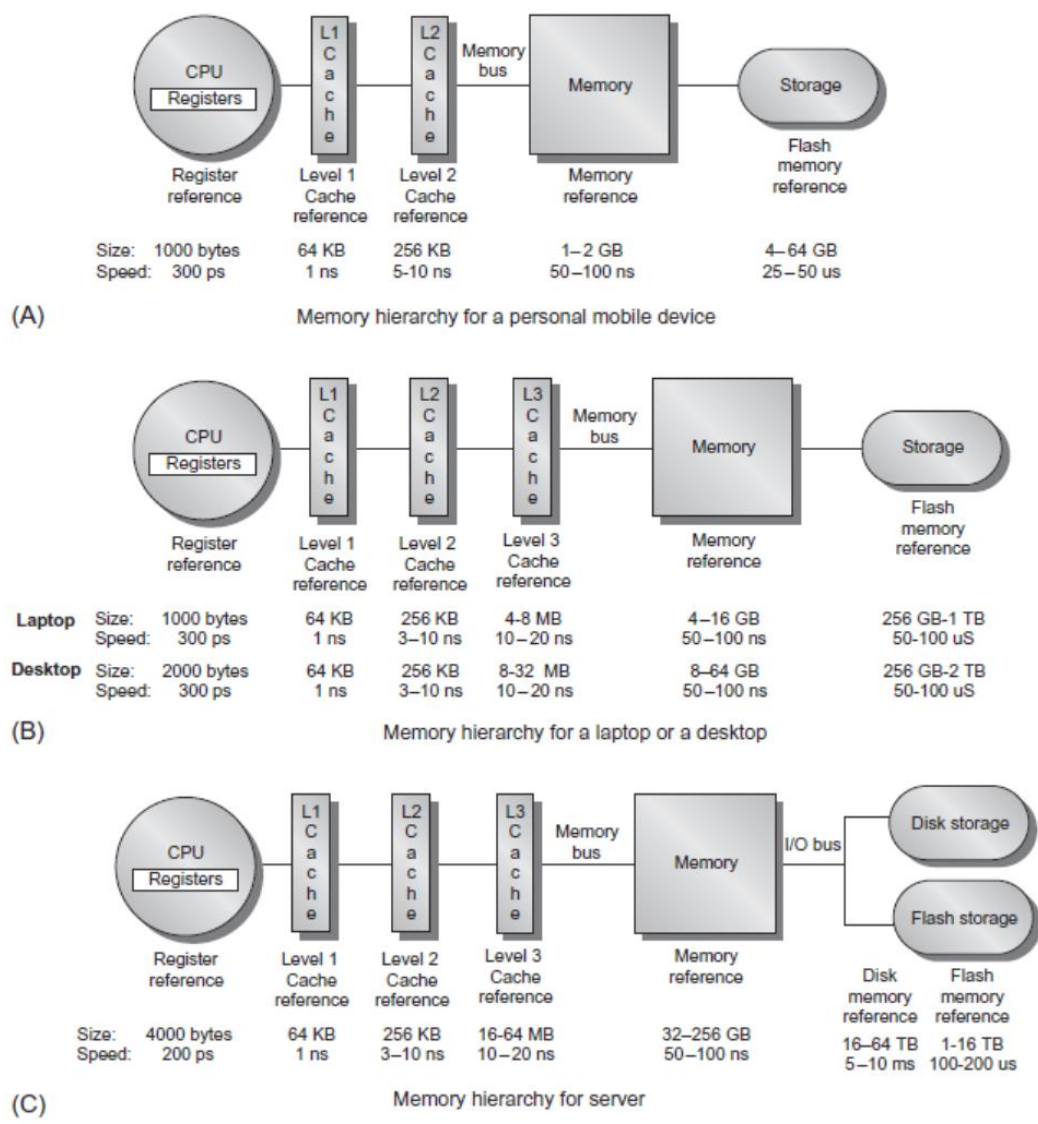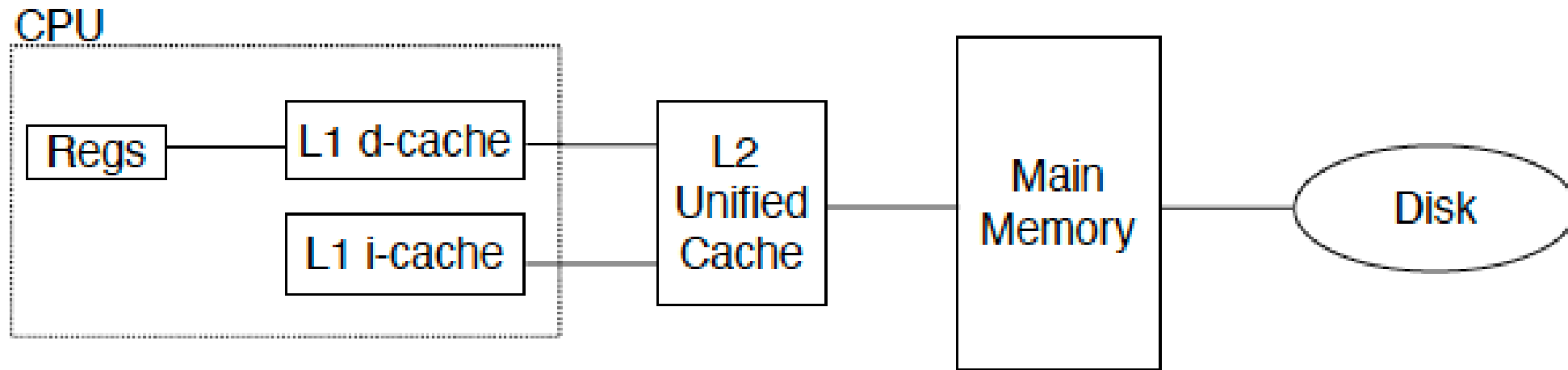|		
Cache	full associative	write allocate
Virtual memory	dirty bit	unified cache
Memory stall cycles	block	block offset
misses per instruction	direct mapped	write back
Valid bit	data cache	locality
Block address	hit time	address trace
Write through	cache miss	set
Instruction cache	page fault	miss rate
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?

Smaller

Bigger

fast

slow

Registers

L1-Cache
(On-Chip)

L2-Cache   (SRAM)

Main Memory (DRAM)

Disk, Tape, ect.

(A) Memory hierarchy for a personal mobile device

	Register reference	Level 1 Cache reference	Level 2 Cache reference	Memory reference	Flash memory reference
Size:	1000 bytes	64 KB	256 KB	1–2 GB	4–64 GB
Speed:	300 ps	1 ns	5-10 ns	50–100 ns	25–50 us

(B) Memory hierarchy for a laptop or a desktop

		Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference	Memory reference	Flash memory reference
Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4–16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS

(C) Memory hierarchy for server

	Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference	Memory reference	Disk memory reference	Flash memory reference
Size:	4000 bytes	64 KB	256 KB	16-64 MB	32–256 GB	16–64 TB	1-16 TB
Speed:	200 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	5–10 ms	100-200 us

A typical multi-level cache organization

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).

Proc

Unified Cache-1

Unified Cache-2

I-Cache-1	Proc	D-Cache-1
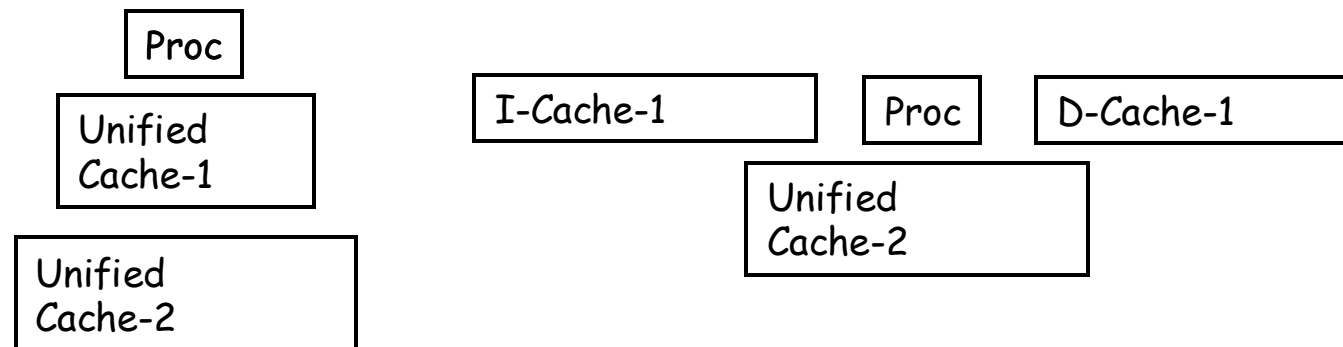
Unified Cache-2

Cache type	Associativity (E)	Block size (B)	Sets (S)	Cache size (C)
on-chip L1 i-cache	4	32 B	128	16 KB
on-chip L1 d-cache	4	32 B	128	16 KB
off-chip L2 unified cache	4	32 B	1024–16384	128 KB–2 MB

Intel Pentium cache organization

The cache just before and just after a reference to a word Xn that is not initially in the cache.

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_3

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_n
X_3

a. Before the reference to X_n b. After the reference to X_n

This reference causes a miss that forces the cache to fetch Xn from memory and insert into the cache.

How do we know if a data item is in the cache?
Moreover, if it is, how do we find it?

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
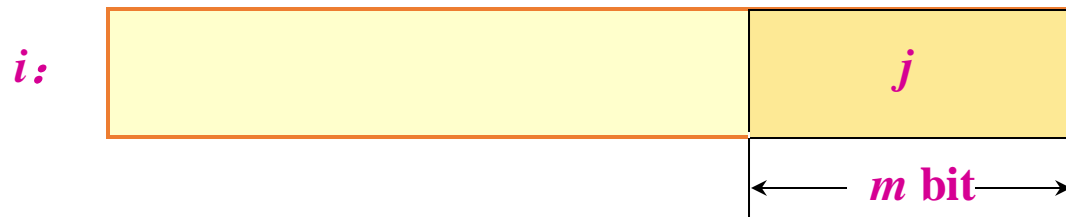
Direct mapped

(Block address) modulo (Number of blocks in the cache)

$i:$

j

m **bit**

A cache structure in which each memory location is mapped to exactly one location in the cache.

Cache

Memory

0
1
2
3
4
5
6
7

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Q1: Block Placement

Direct mapped

(Block address) modulo (Number of blocks in the cache)

i :

j

m **bit**

A cache structure in which each memory location is mapped to exactly one location in the cache.

Cache

000 001 010 011 100 101 110 111

00001 00101 01001 01101 10001 10101 11001 11101
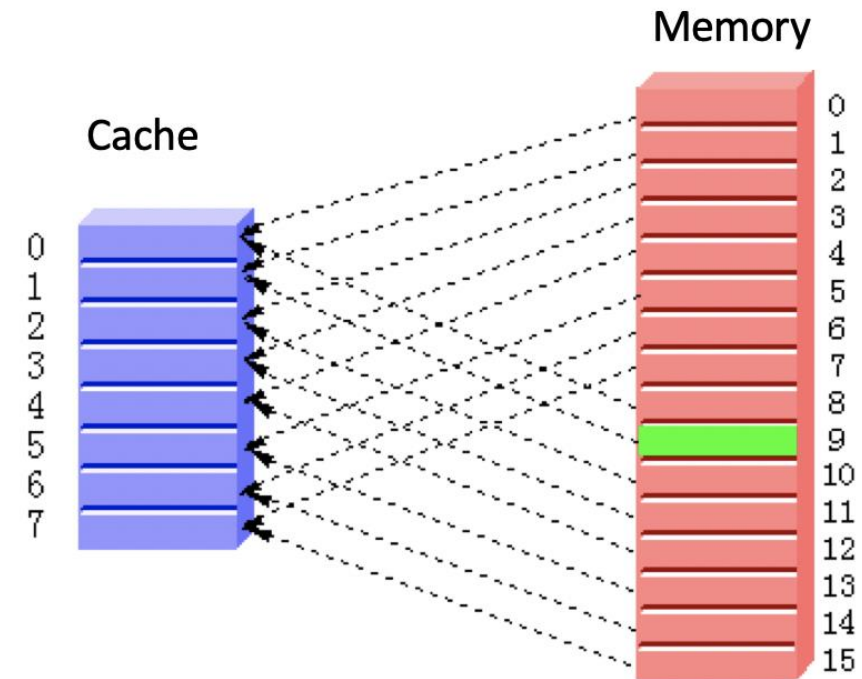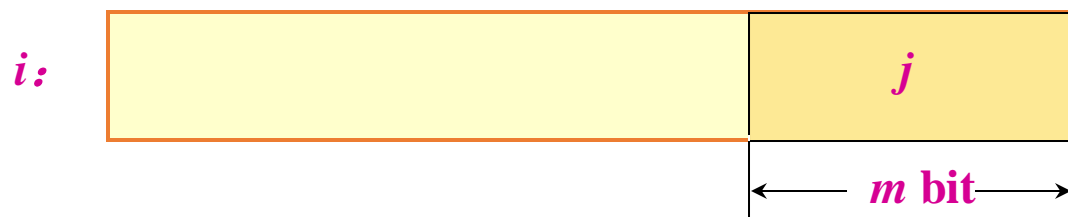
Memory
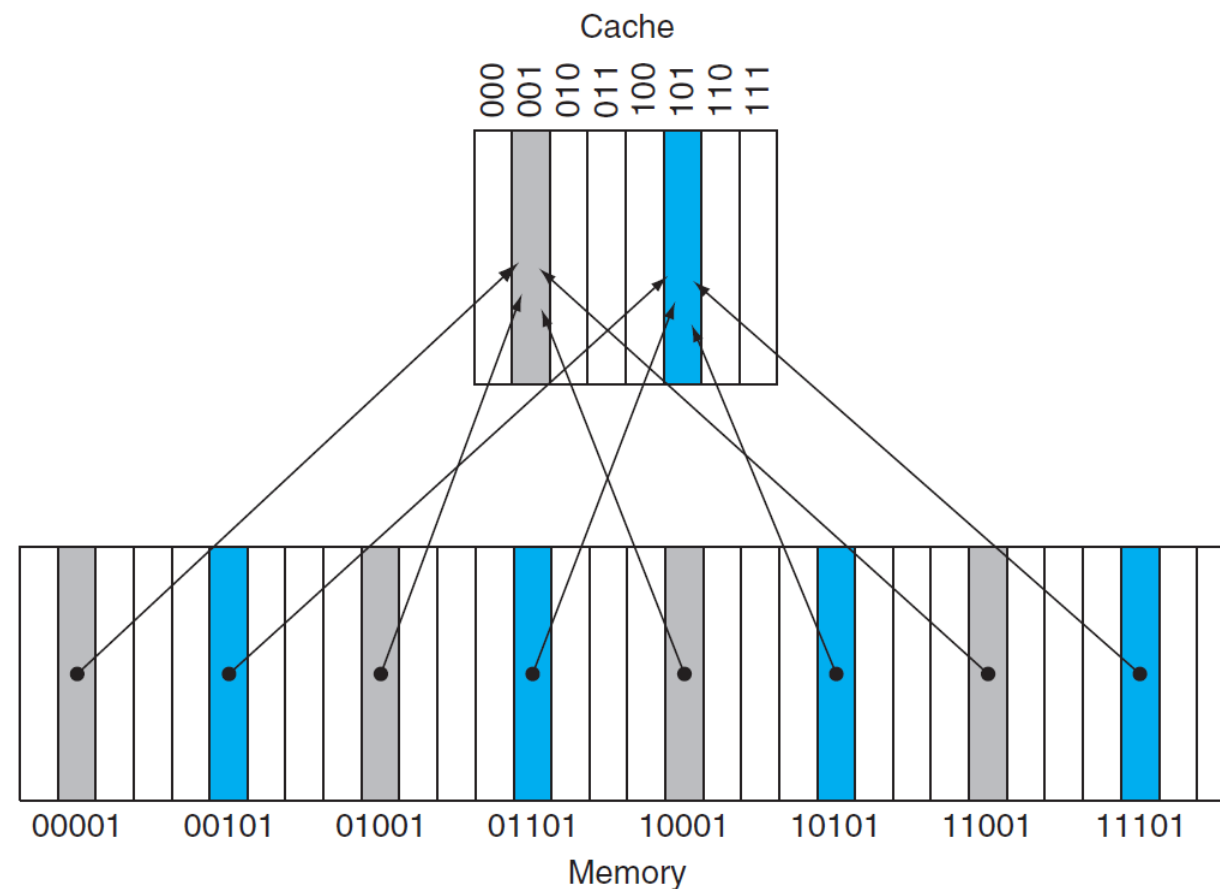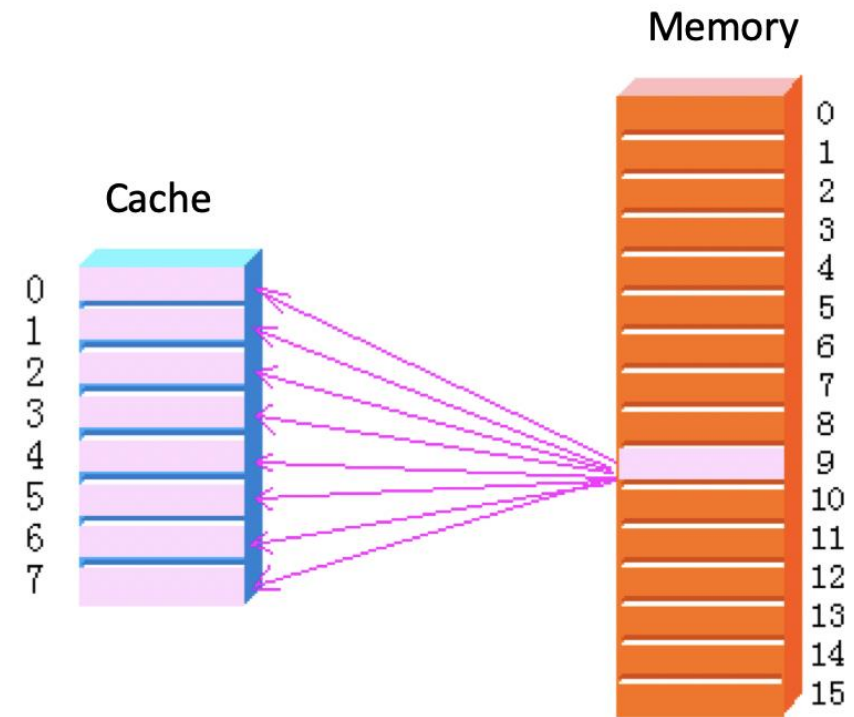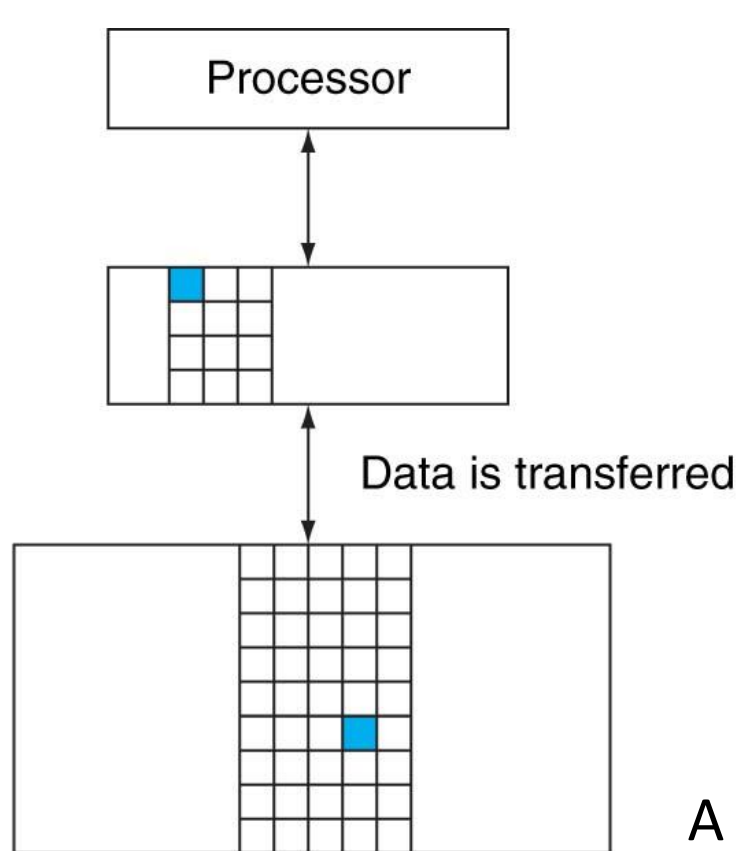
Fully-associative

A cache structure in which a block can be placed in any location in the cache.

2-way Set-associative

$i:$

k

g bit
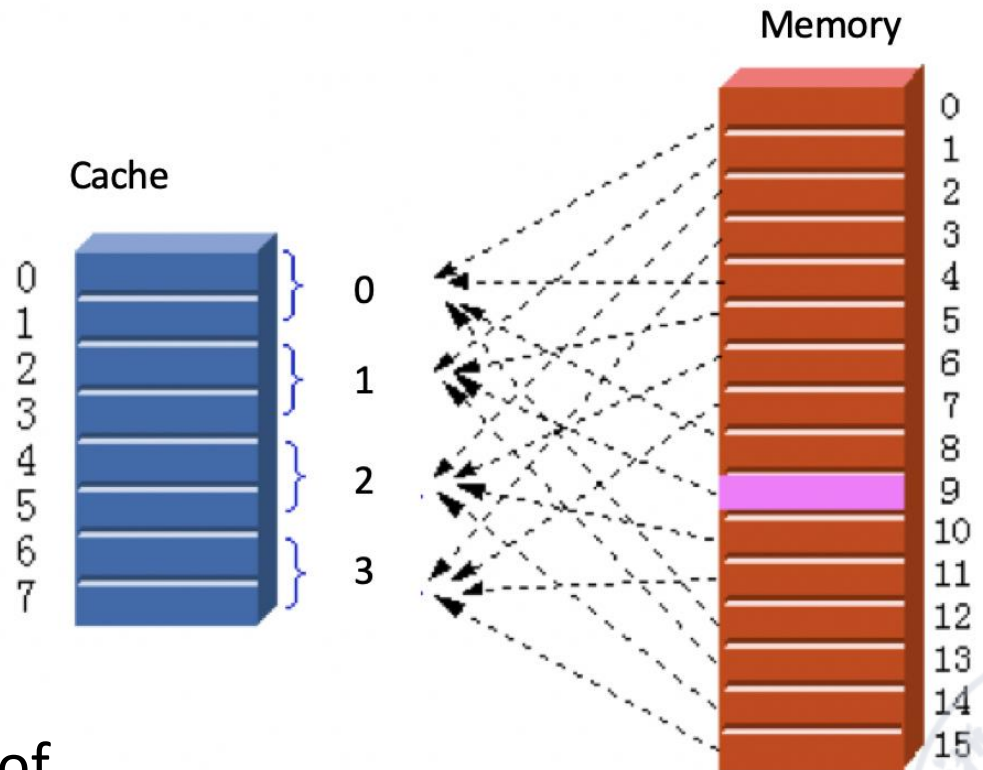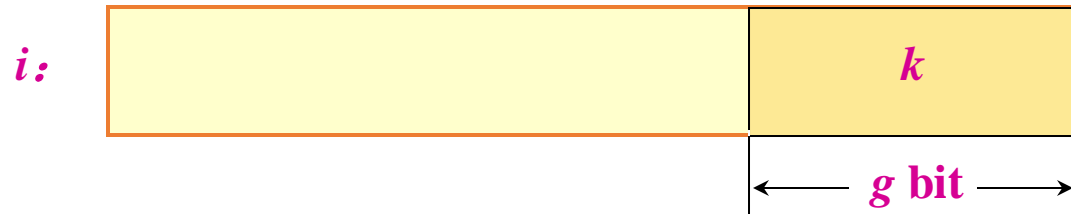
A cache that has a fixed number of locations (at least two) where each block can be placed.

4-way Set-associative
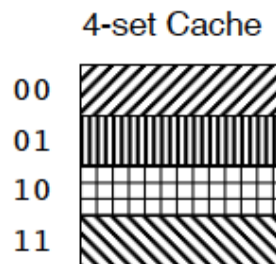
Why Index With the Middle Bits?

If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set.
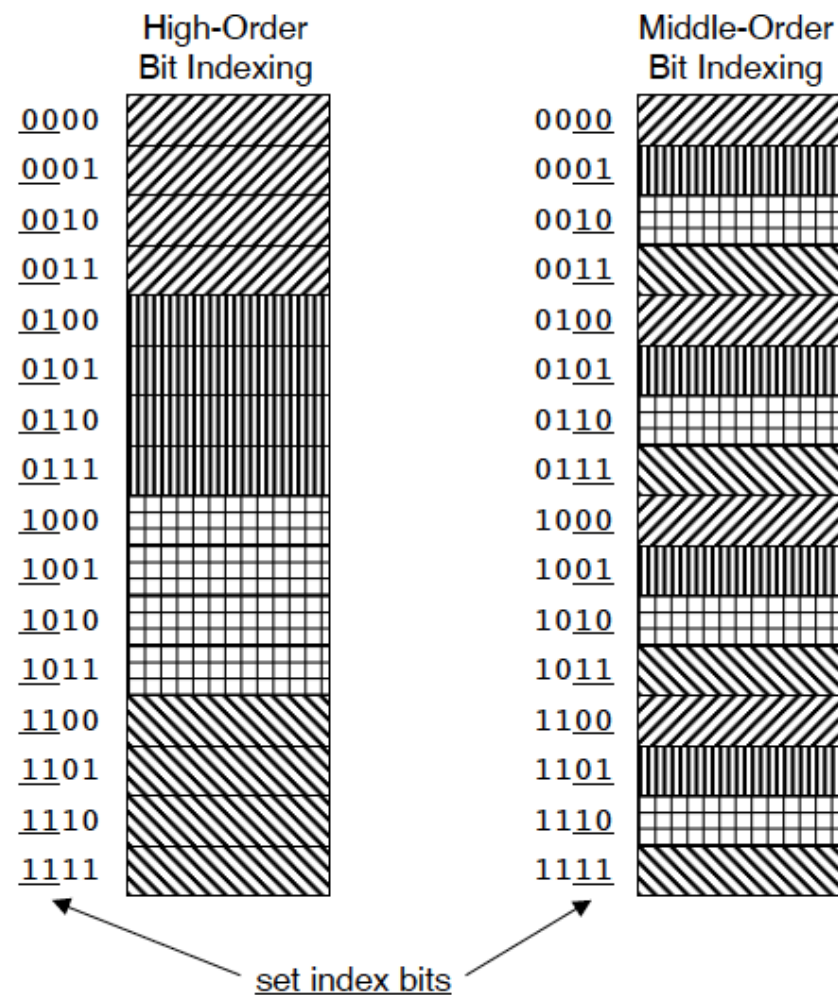
Q1: Block Placement

- Direct mapped
  - Block can only go in one place in the cache.

- Fully

  •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 as

  - 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.

n sets => *n-way set associative*

Direct-mapped cache => one block per set

Fully associative => one set

8-32 Block Placement

Direct Mapped
block 12 can go only into
block 4 (12 mod 8)

Fully-associative
block 12 can go anywhere

2-way Set-associative
block 12 can go anywhere in set 0
(12 mod 4)
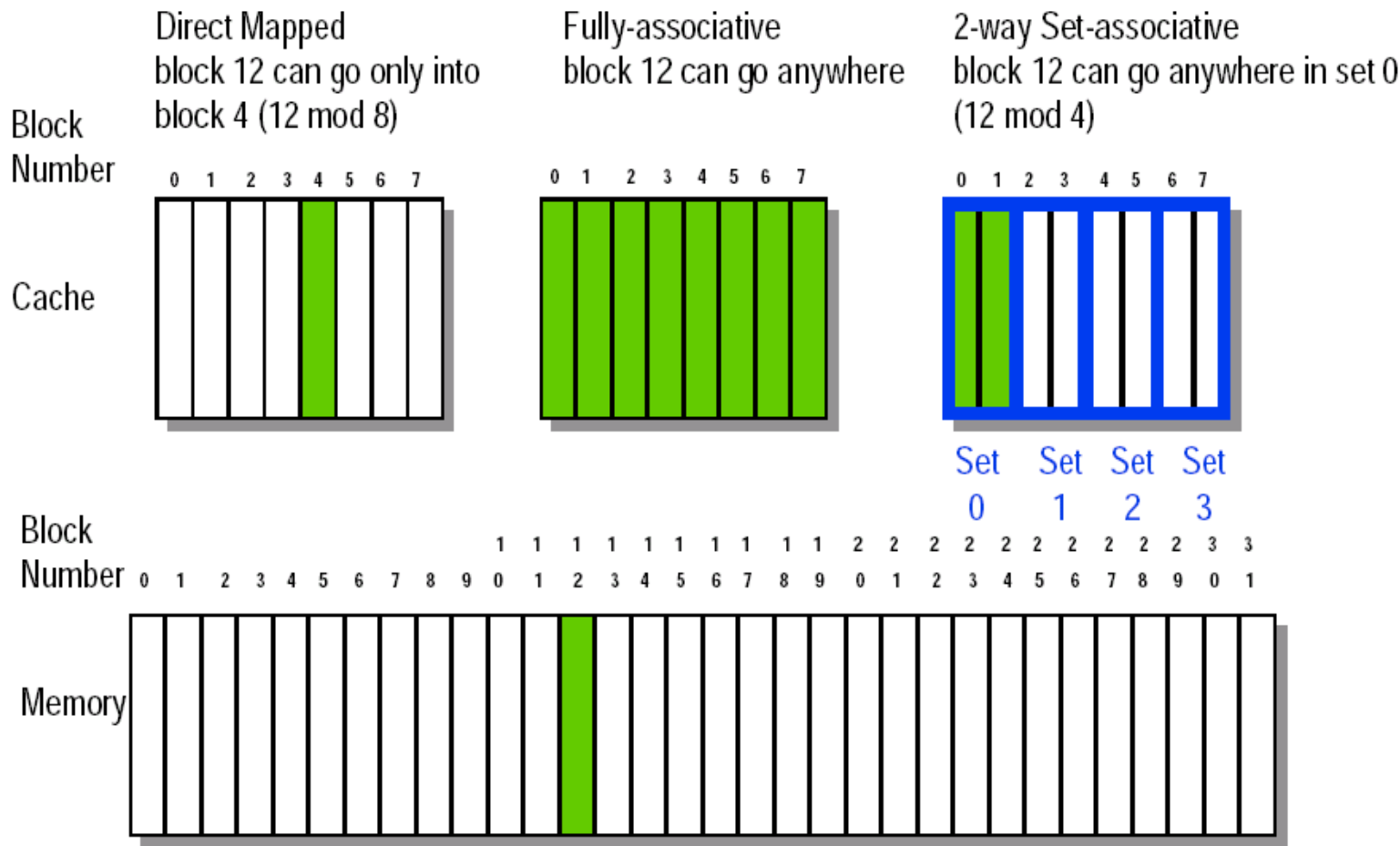
N-way Set-associative

- 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?

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)

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
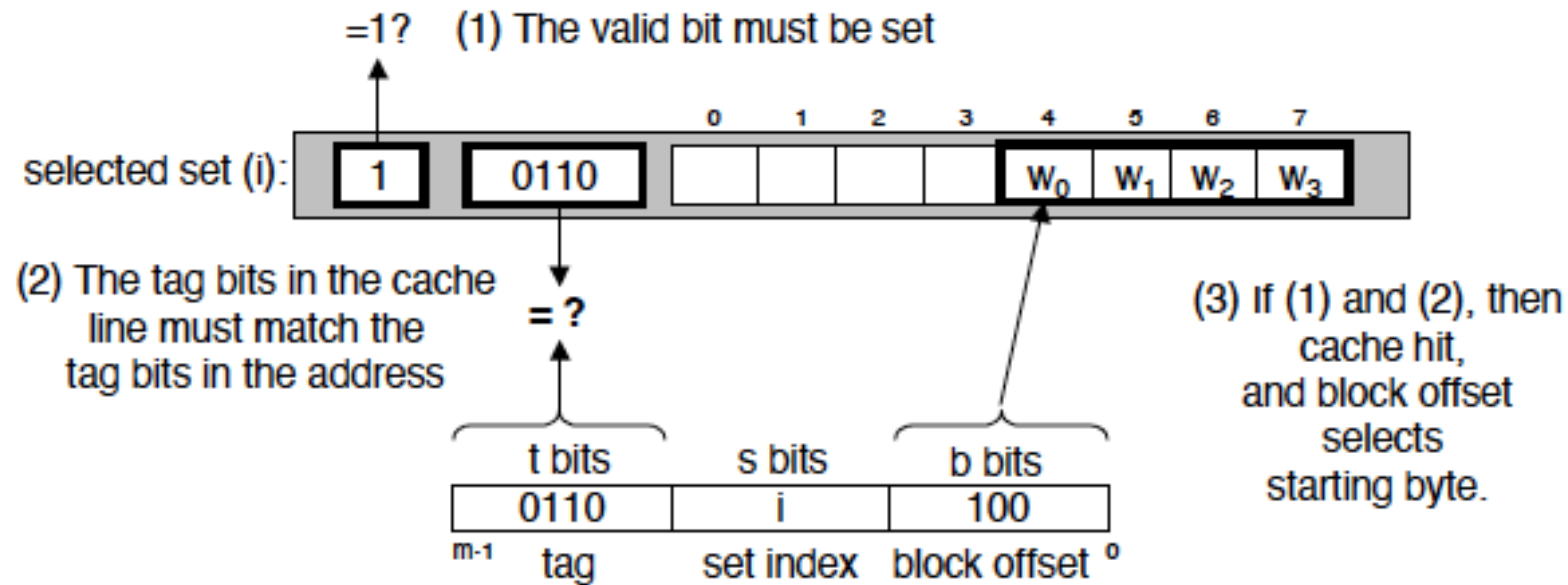
The Format of the Physical Address

- 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 log2(#sets) for set-associative caches, or log2(#blocks) for direct-mapped caches

Block Address

Tag | Index | Offset

Stored in cache and used in comparison with CPU address | Selects set | Selects data within the block

- The Byte Offset field selects
  - The byte within the block
  - Has as many bits as log2(size of block)

TAG | Index | Byte Offset

- 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

Line Matching and Word Selection

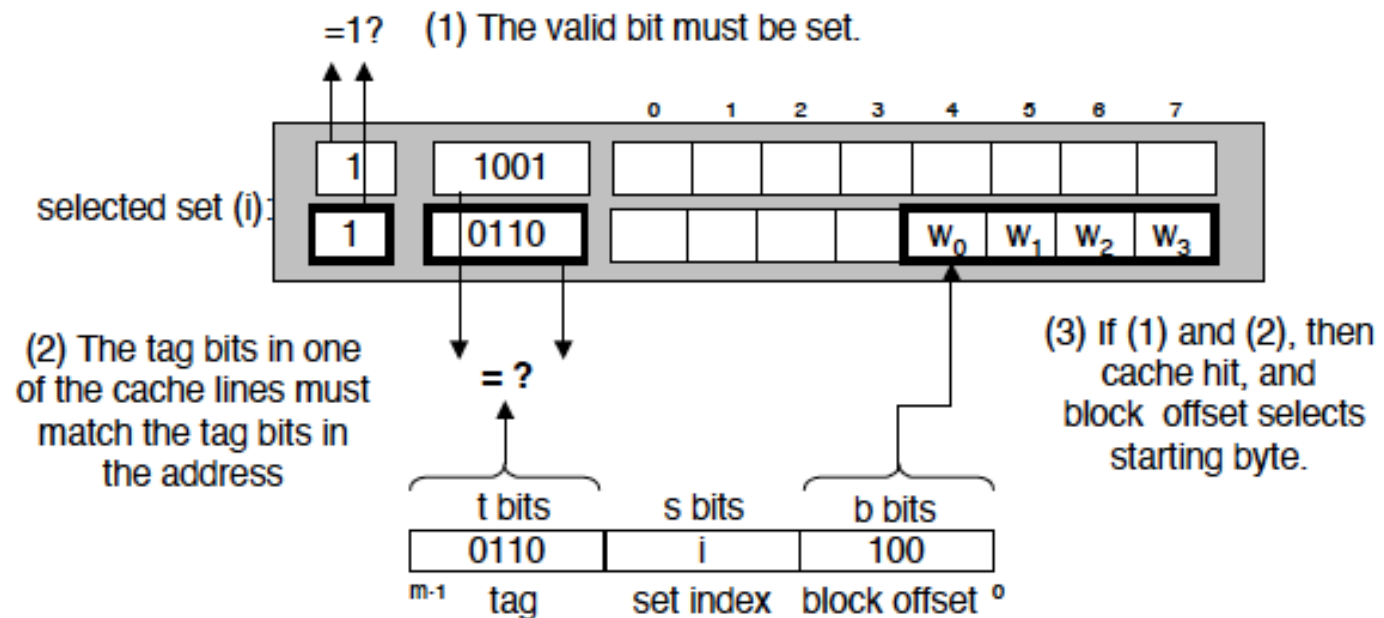=1? (1) The valid bit must be set

selected set (i):

(2) The tag bits in the cache
line must match the
tag bits in the address

=?

(3) If (1) and (2), then
cache hit,
and block offset
selects
starting byte.

t bits	s bits	b bits
0110	i	100
tag	set index	block offset
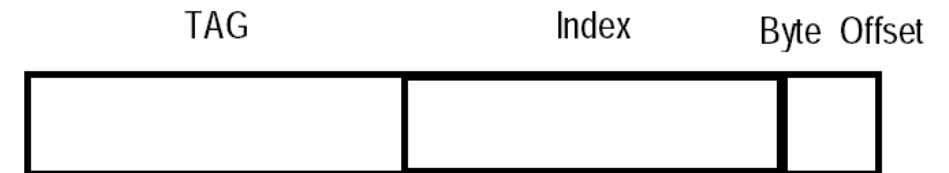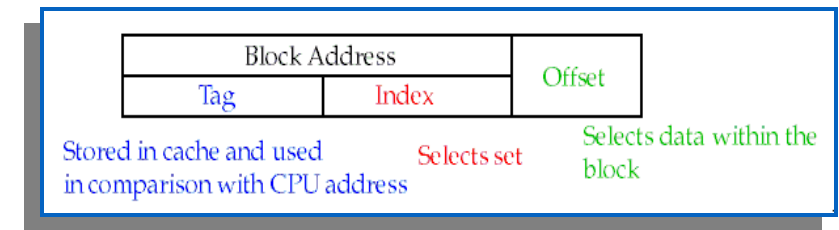
direct-mapped cache

Direct-mapped Cache Example (1-word Blocks)

Line Matching and Word Selection

=1? (1) The valid bit must be set.

selected set (i):

(2) The tag bits in one of the cache lines must match the tag bits in the address

= ?

(3) If (1) and (2), then cache hit, and block offset selects starting byte.

t bits	s bits	b bits
0110	i	100
tag	set index	block offset
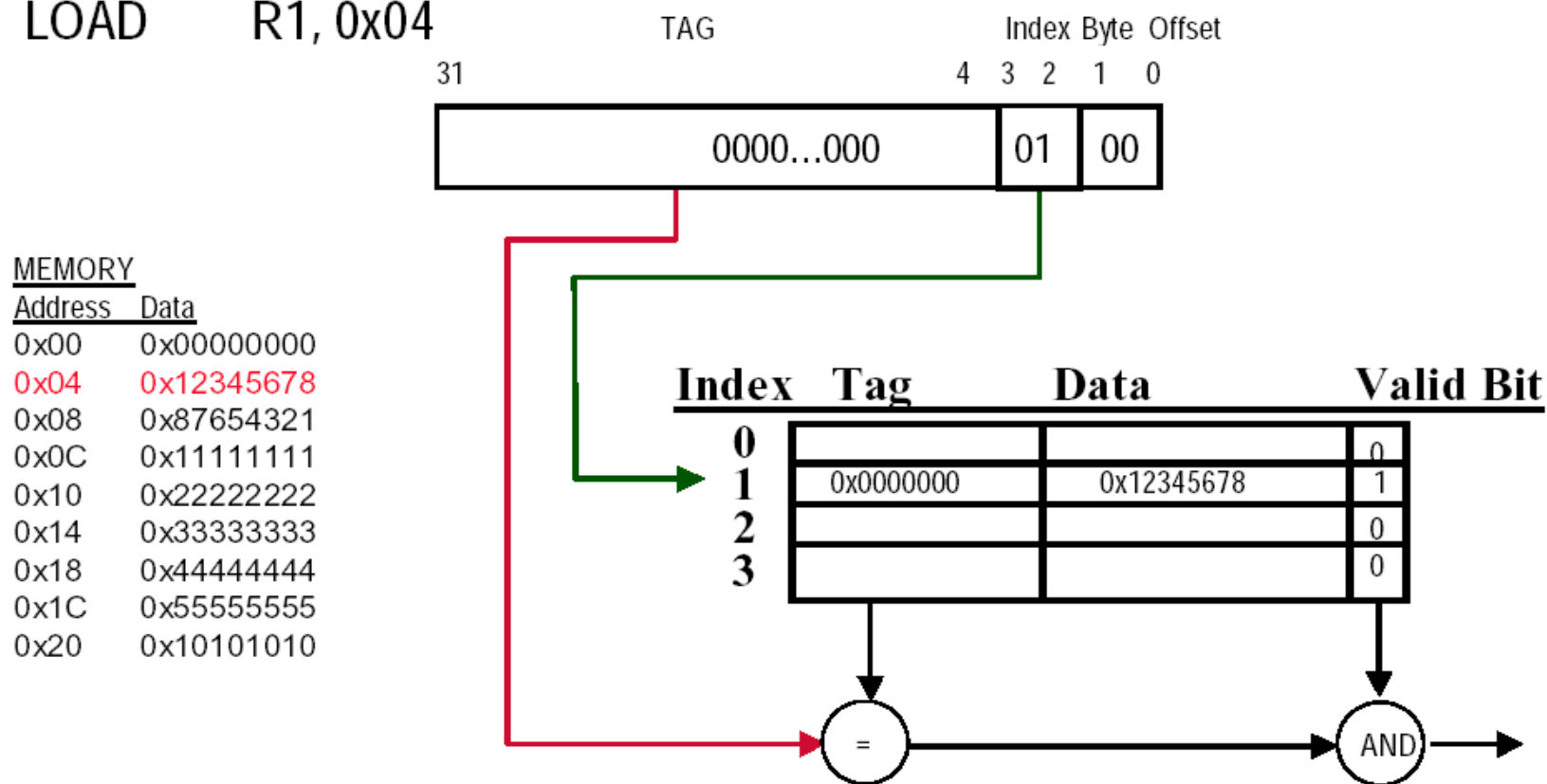
set associative cache

Fully-Associative Cache example (1-word Blocks)

- Assume cache has 4 blocks
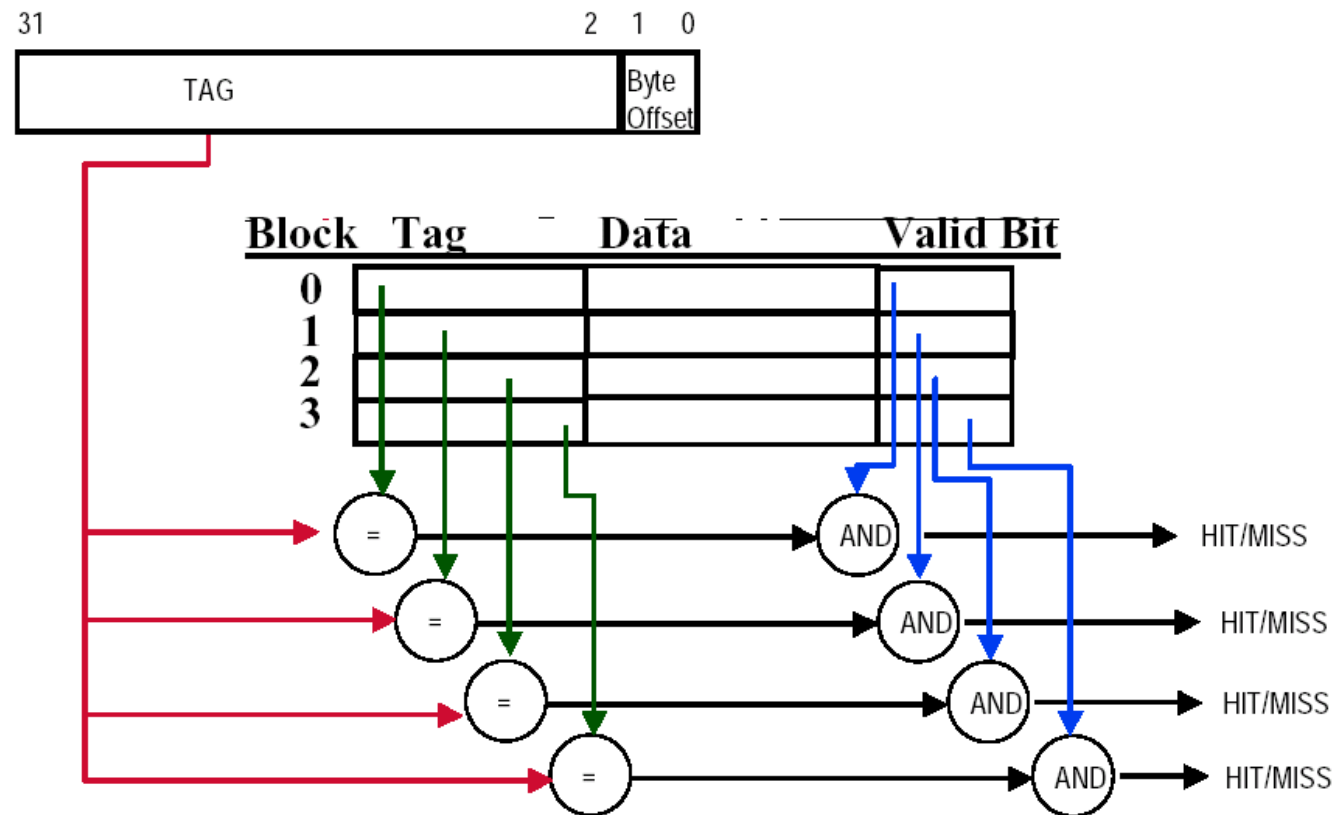
Line Matching and Word Selection
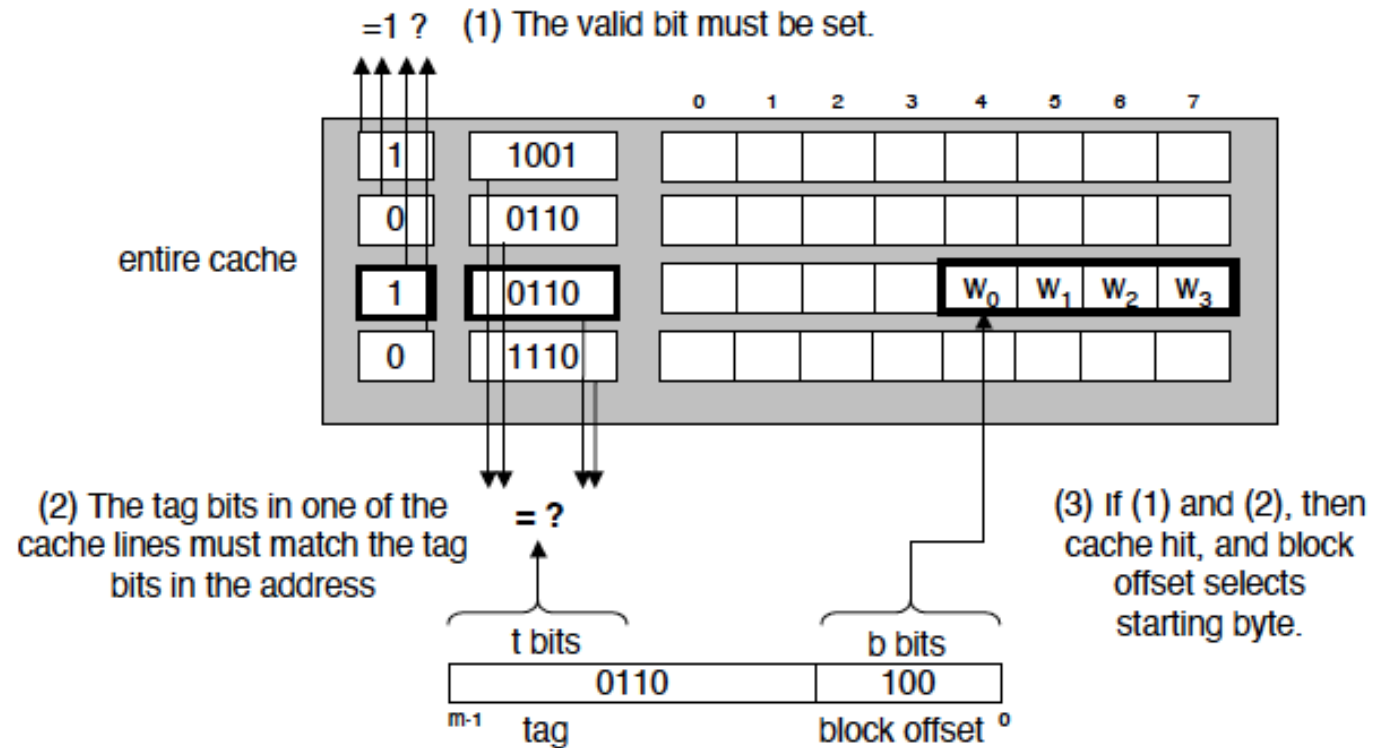
fully associative cache

2-Way Set-Associative Cache

- Assume cache has 4 blocks and each block is 1 word
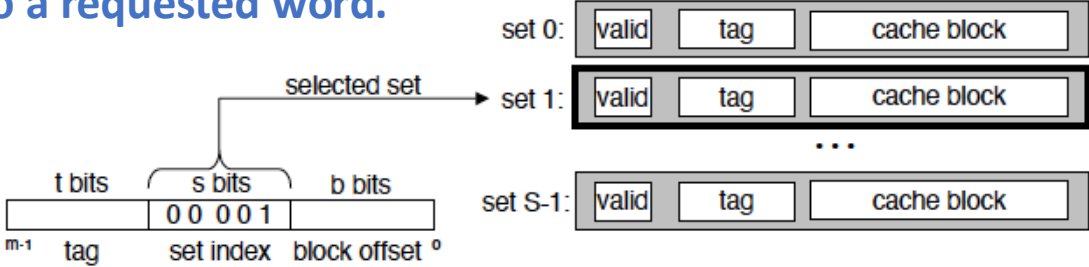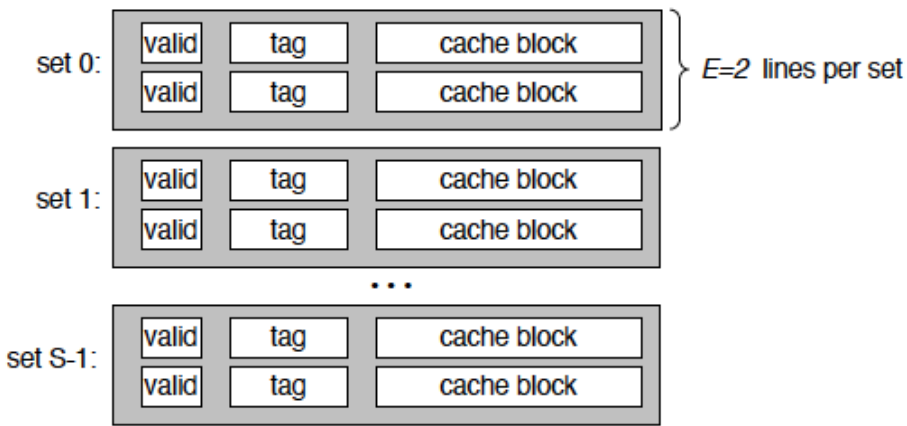- 2 blocks per set, hence 2 sets per cache
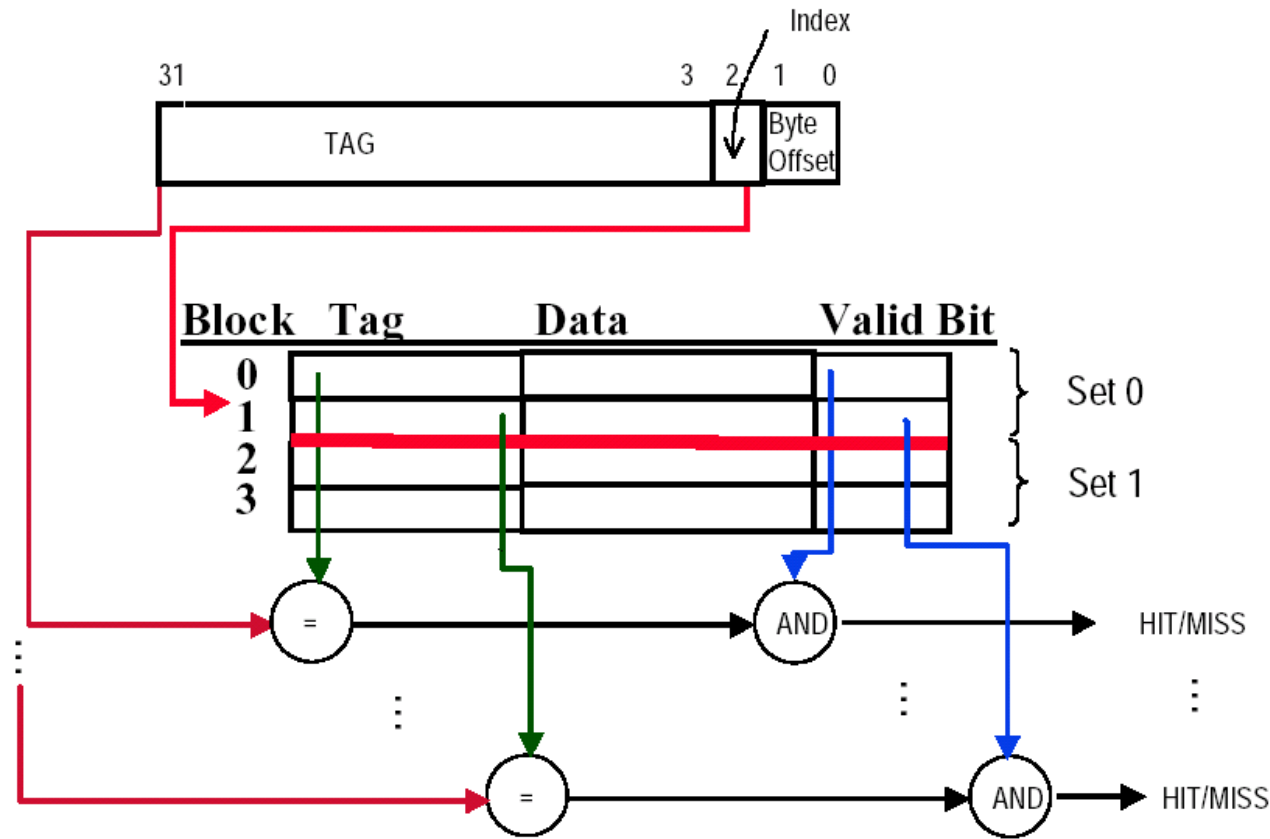
tag: A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.
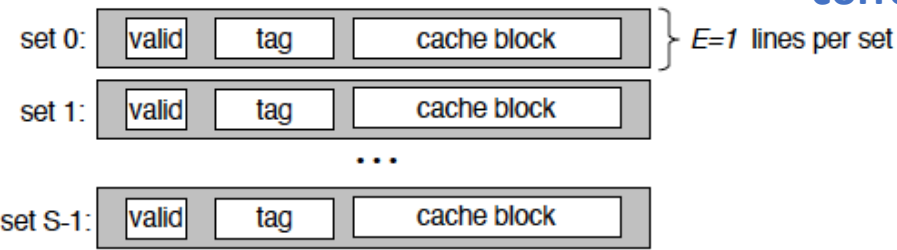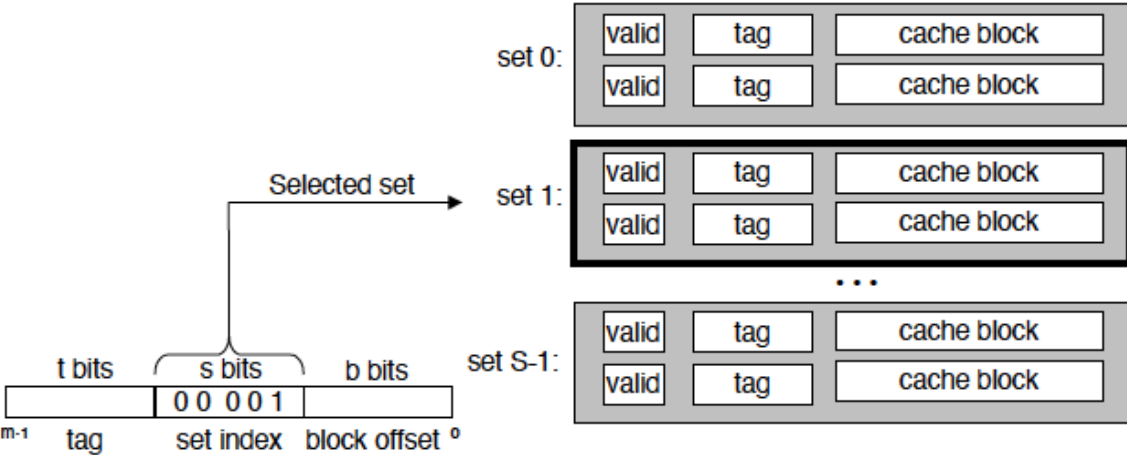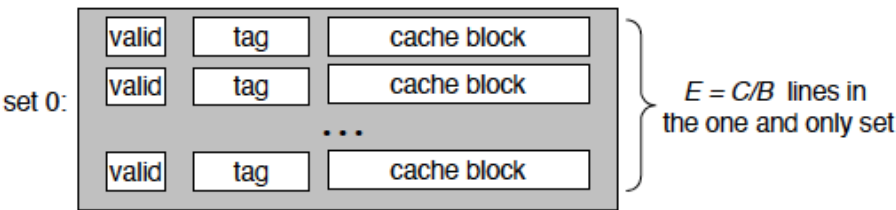

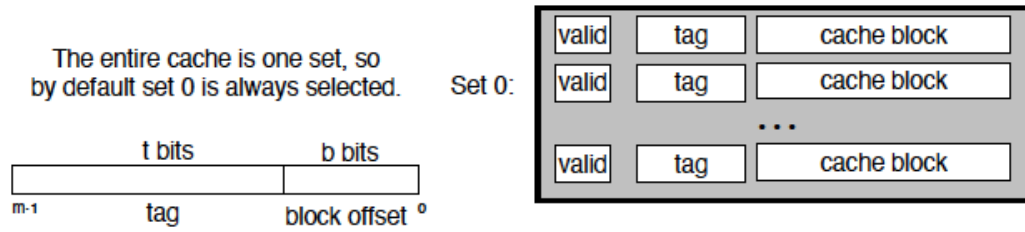*Direct-mapped cache*

Set selection in a direct-mapped cache

Set associative cache

Set selection in a set associative cache

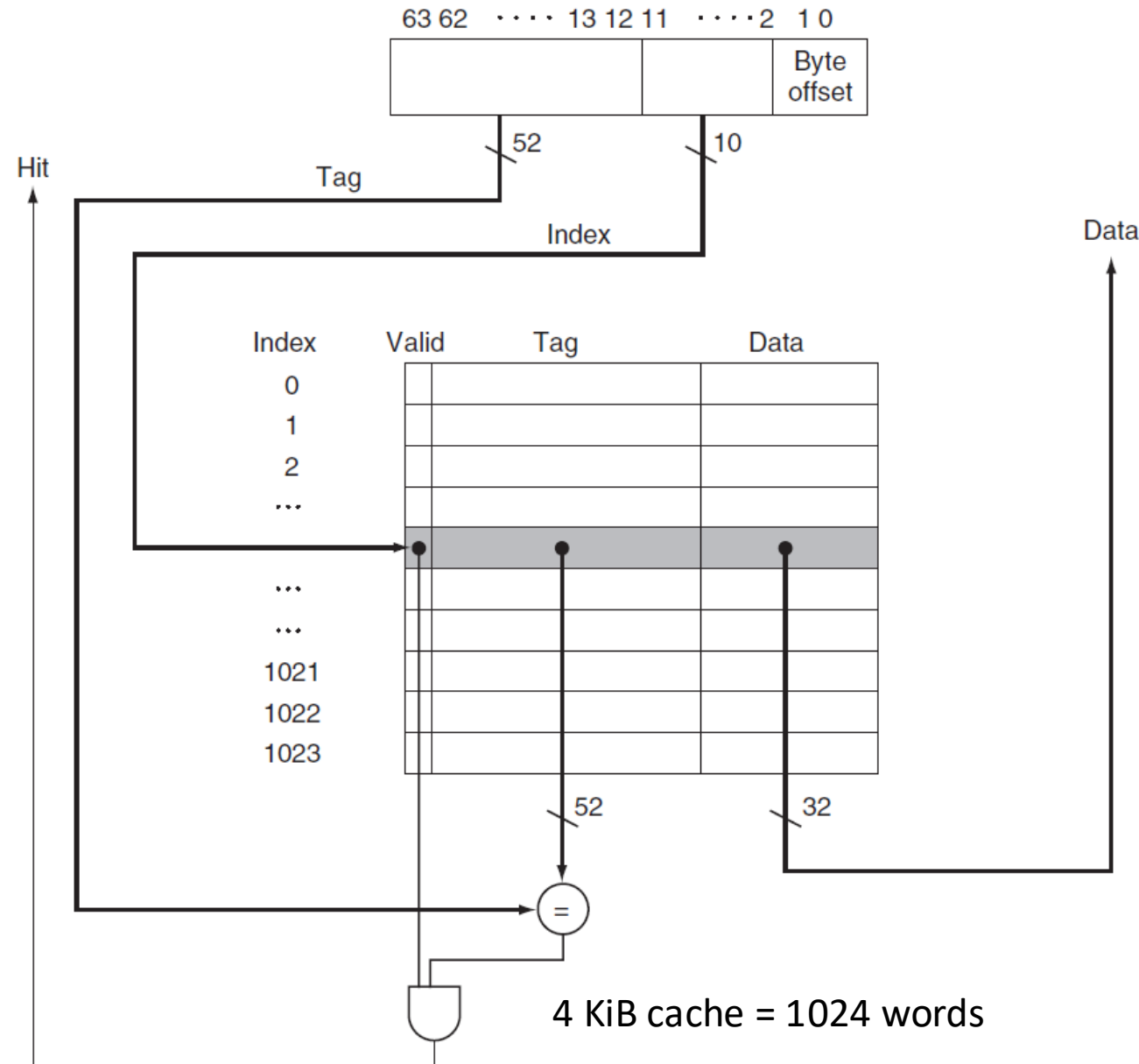*Fully set associative cache*

Set selection in a fully associative cache

4 KiB cache = 1024 words

The size of the block was one word (4 bytes).

- 64-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so n bits are used for the index
- The block size is
  - ✓ 2^m words (2^{m+2} bytes = 2^{m+5} bits)
  - ✓ m bits are used for the word within the block, and two bits are used for the byte part of the address

tag $64 - (n + m + 2)$

cache $2^n \times$ (block size+ tag size + Valid size).

$= 2^n \times (2^m \times 32 + (64 - n - m - 2) + 1)$
$= 2^n \times (2^m \times 32 + 63 - n - m)$.

actual size/complete cache size

Bits in a Cache

Example 1: How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

Answers:

16 KiB = 4096 words = 2^{12} words

The block size is 4 words = 2^2 words, so m=2

2^{12} / 2^2 = 1024 blocks, so n=10

Each block has 4*32 =128 bits of data plus a tag, plus a valid bit

tag = 64-(n+m+2) = 64-(10+2+2) =50 bits

Cache size = $2^n \times$(block size+ tag size + Valid size)

$\qquad$ = $2^{10} \times$(4*32 + 50 +1)=179 Kib =22.375 KiB

Mapping an Address to a Multiword Cache Block

Example 2: Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

Answers:

The cache block number

= (Block address) modulo (Number of blocks in the cache)

Block address = Byte address/ Bytes per block = 1200 /16 =75

75 modulo 64 =11

In fact, this block maps all addresses between 1200 and 1215.

Using a Finite-State Machine to Control a Simple Cache
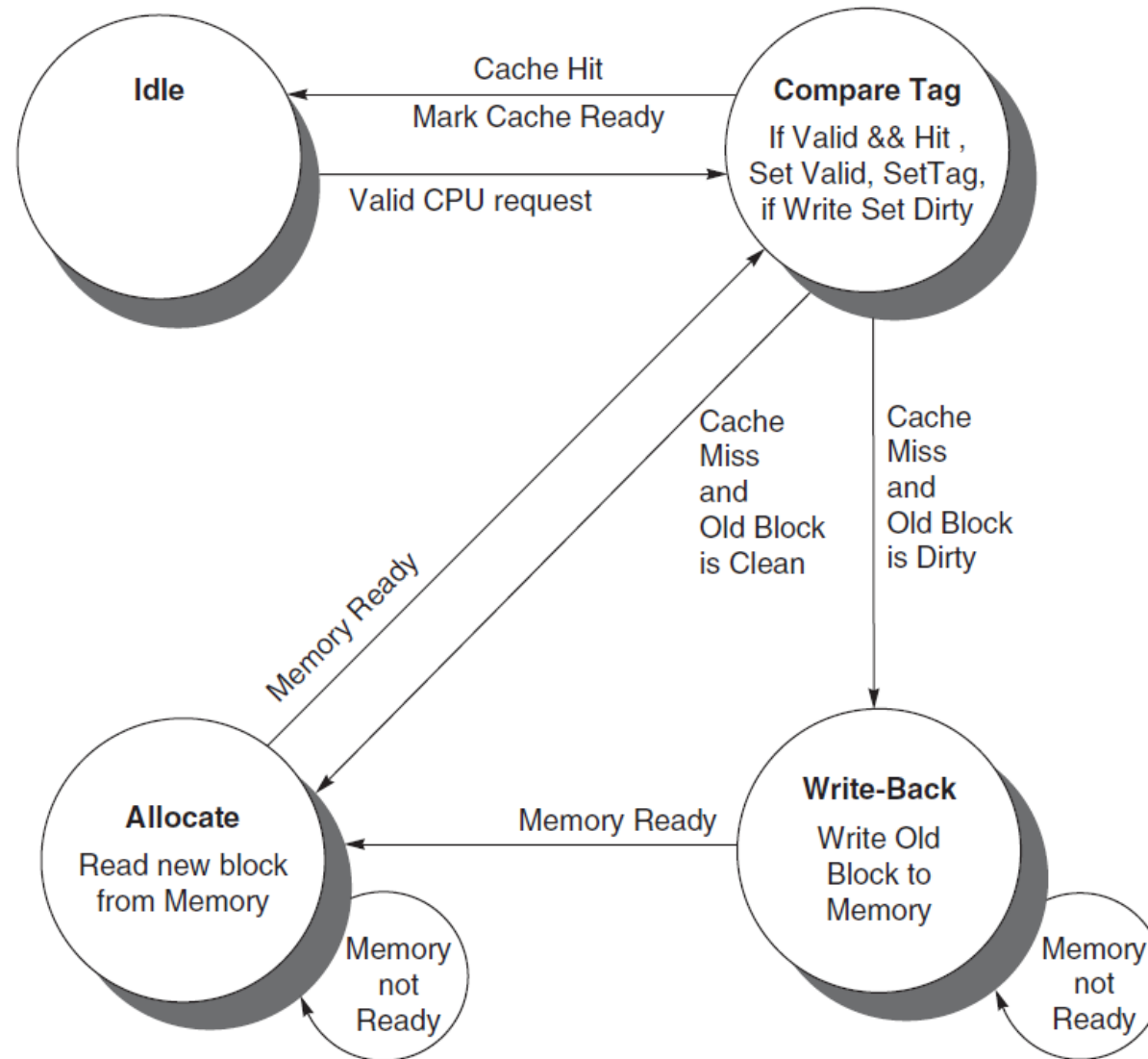
finite-state machine

A sequential logic function consisting of a set of inputs and outputs, next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function

A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.
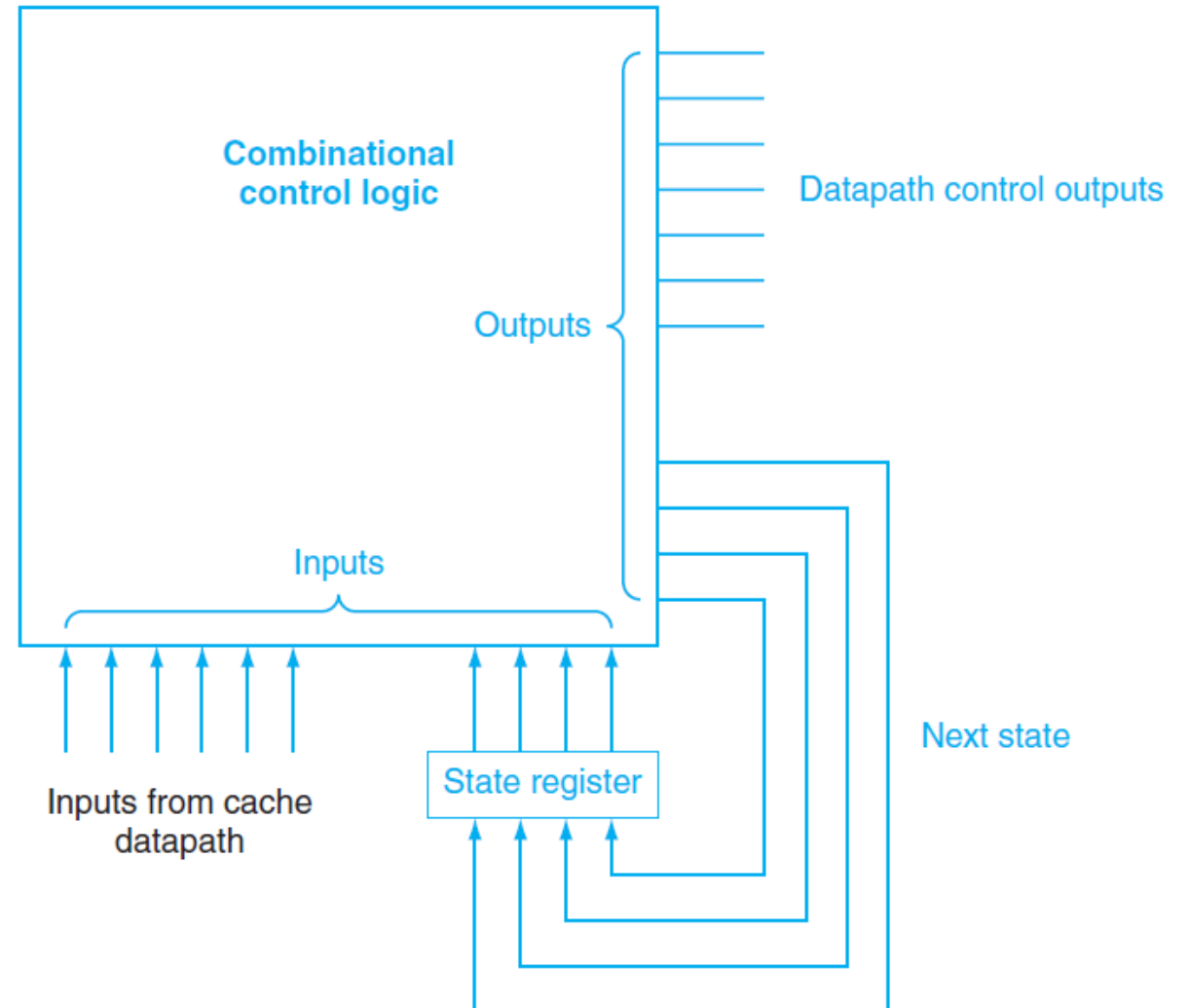
Using a Finite-State Machine to Control a Simple Cache

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)

Q3: Block Replacement

- In a direct-mapped cache, there is only one block that can be replaced

Binary address of reference	Assigned cache block (where found or placed)
10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$
11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$
10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$
11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$
10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$
00011_{two}	$(00011_{two} \bmod 8) = 011_{two}$
10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$
10010_{two}	$(10010_{two} \bmod 8) = 010_{two}$
10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

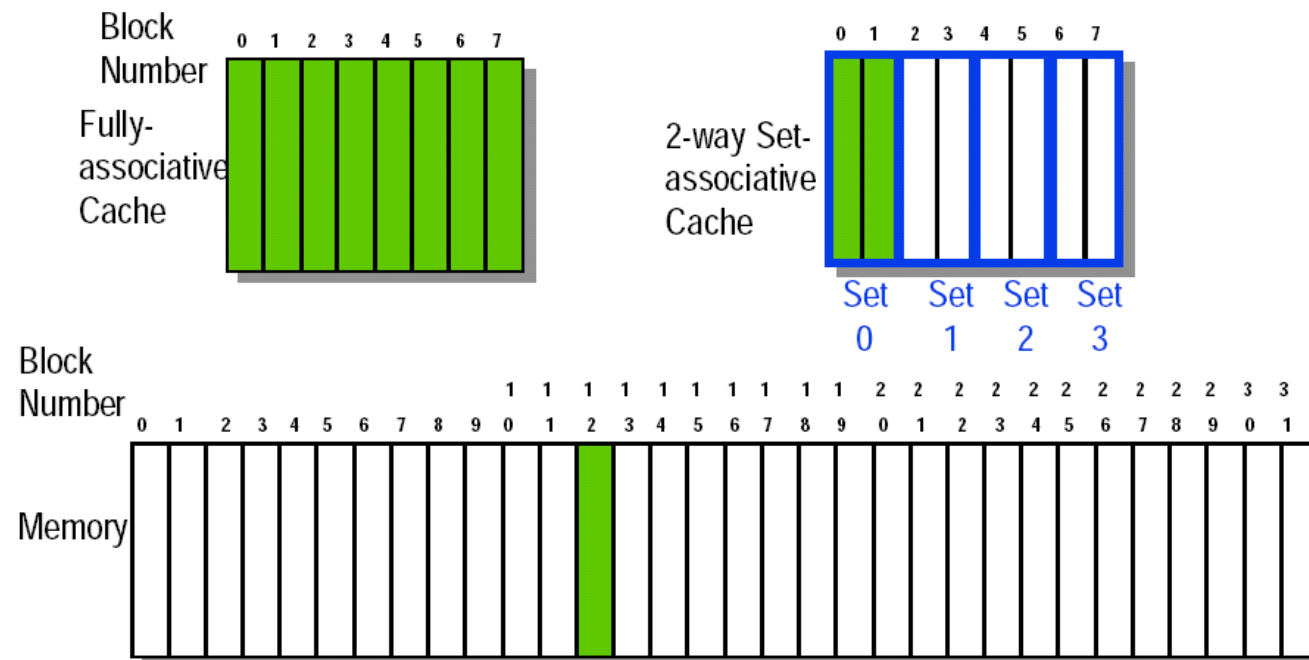| 10110	11010	10110	11010	10000	00011	10000	10010	10000
miss	miss	hit	hit	miss	miss	hit	miss	hit

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*
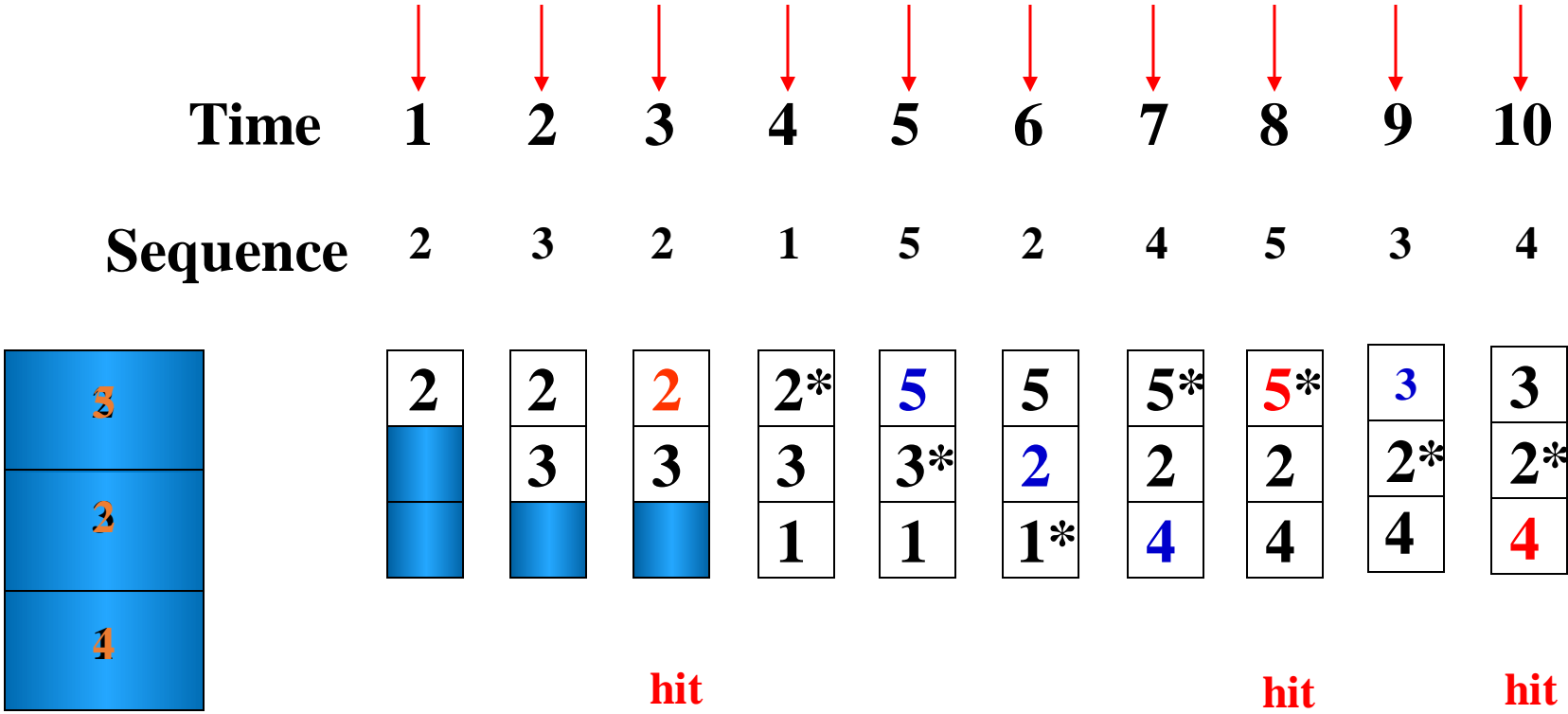
Strategy of Block Replacement

Suppose:

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

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

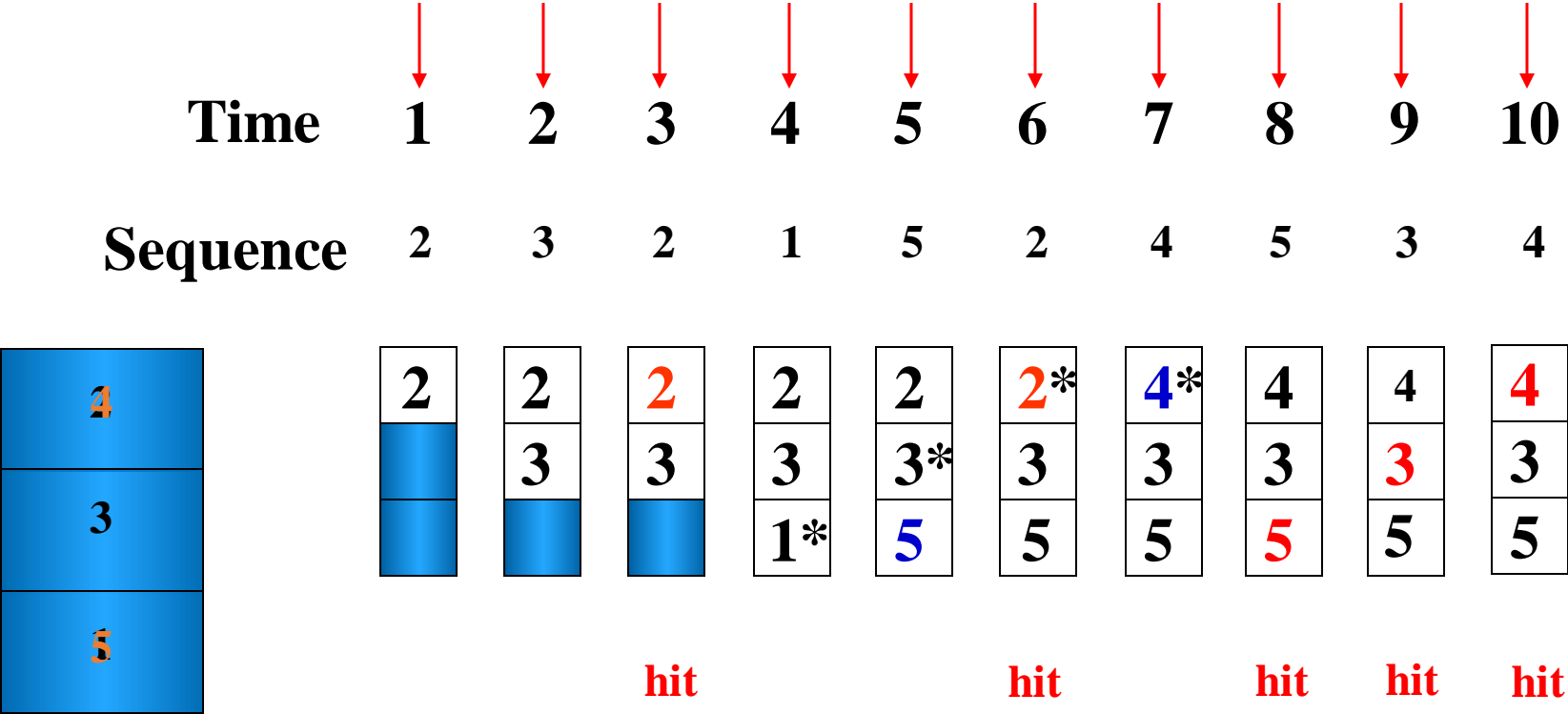- FIFO, LRU and OPT are used to simulate the use and replacement of cache block.

FIFO

LRU

OPT

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

2	2	2	2	2	2*	4*	4	4	4
	3	3	3	3*	3	3	3	3	3
			1*	5	5	5	5	5	5
		hit		hit		hit	hit	hit	

The hit rate is related to the replacement algorithm.

thrashing

Time	1	2	3	4	5	6	7	8
Sequence	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**

FIFO

1	1	1*	4	4	4*	3	3
	2	2	2*	1	1	1*	4
		3	3	3*	2	2	2*

n=3

No hit

LRU

n=4

1	1	1*	4	4	4*	3	3
	2	2	2*	2	2	2*	2
		3	3	3*	3	3	2*
			4	4	4	4	4

Hit 4 times
No hit

OPT

1	1	1	1	1*	1	1	1
	2	2	2	2	2*	3*	3
		3*	4*	4	4	4	4*

Hit 3 times

➤ Hit rate is related to access sequence.
Hit rate is related to cache block size.

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)

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)

1	1	1*	4	4	4*	5	5	5*	3	3	3*
	2	2	2*	1	1	1*	1	1	1*	4	4
		3	3	3*	2	2	2*	2	2	2*	5

LRU n+1=4

Bt(n+1)

1	1	1	1*	1	1	1	1	1	1	1*	5
	2	2	2	2*	2	2	2	2	2	2	2*
		3	3	3	3*	5	5	5	5*	4	4
			4	4	4	4*	4*	4*	3	3	3

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

			1*	4	4	4*	5	5	5*	3	3	3*
1	1	2	2*	1	1	1*	1	1	1*	4	4
	2	3	3	3*	2	2	2*	2	2	2*	5

Hit Hit

LRU n=4
Hit 4 times

1	1	1	1*	1	1	1	1	1	1	1*	5
	2	2	2	2*	2	2	2	2	2	2	2*
		3	3	3	3*	5	5	5	5*	4	4
			4	4	4	4*	4*	4*	3	3	3

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:

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

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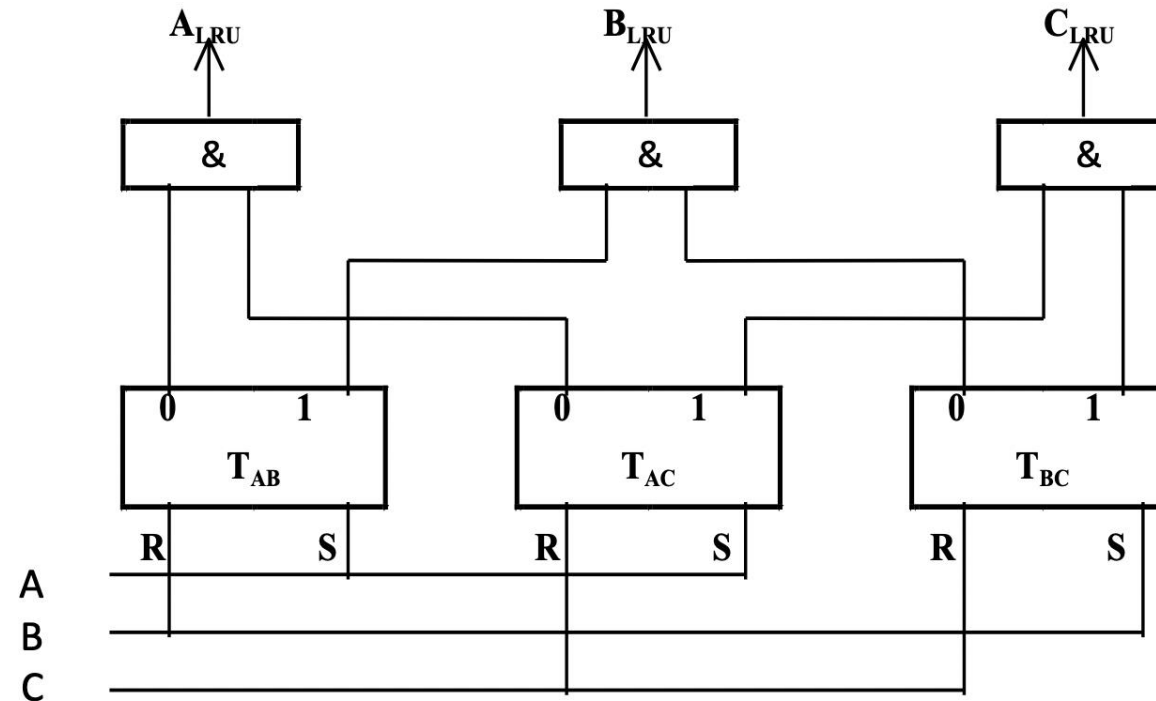
- Change the state of the flip-flop after each access.
  - After accessing block A: $T_{AB}=1$, $T_{AC}=1$
  - After accessing block B: $T_{AB}=0$, $T_{BC}=1$
  - After accessing block C: $T_{AC}=0$, $T_{BC}=0$

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·(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

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)

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
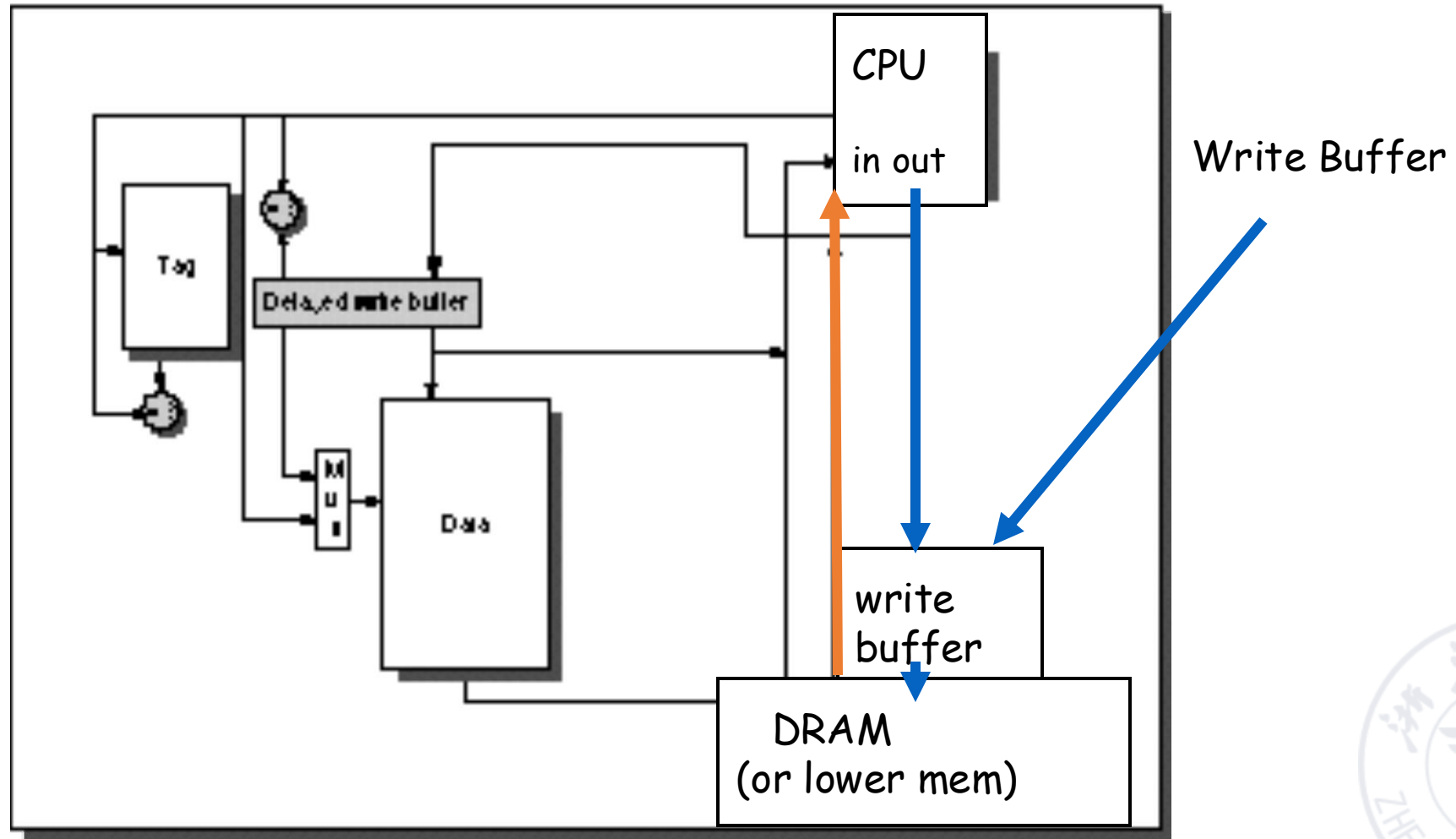
Write misses

- 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 no write-allocate.
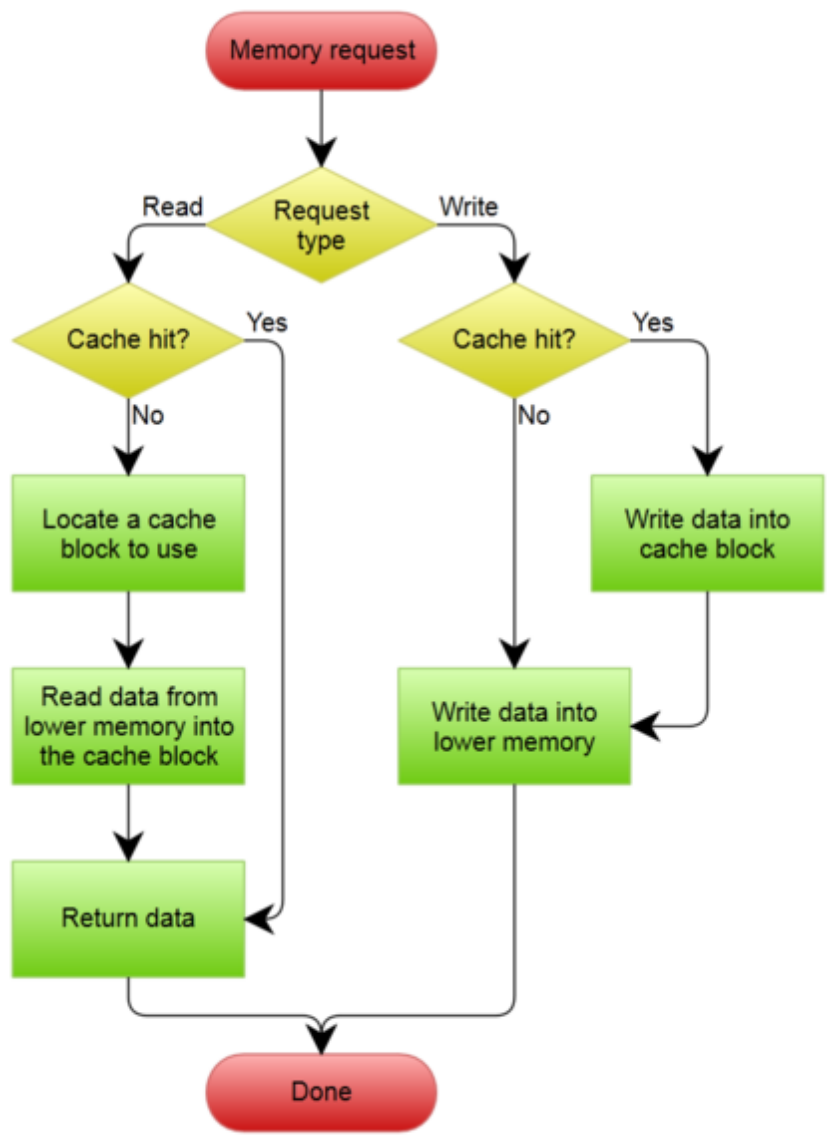
A Write-Through cache with No-Write Allocation
Read allocate
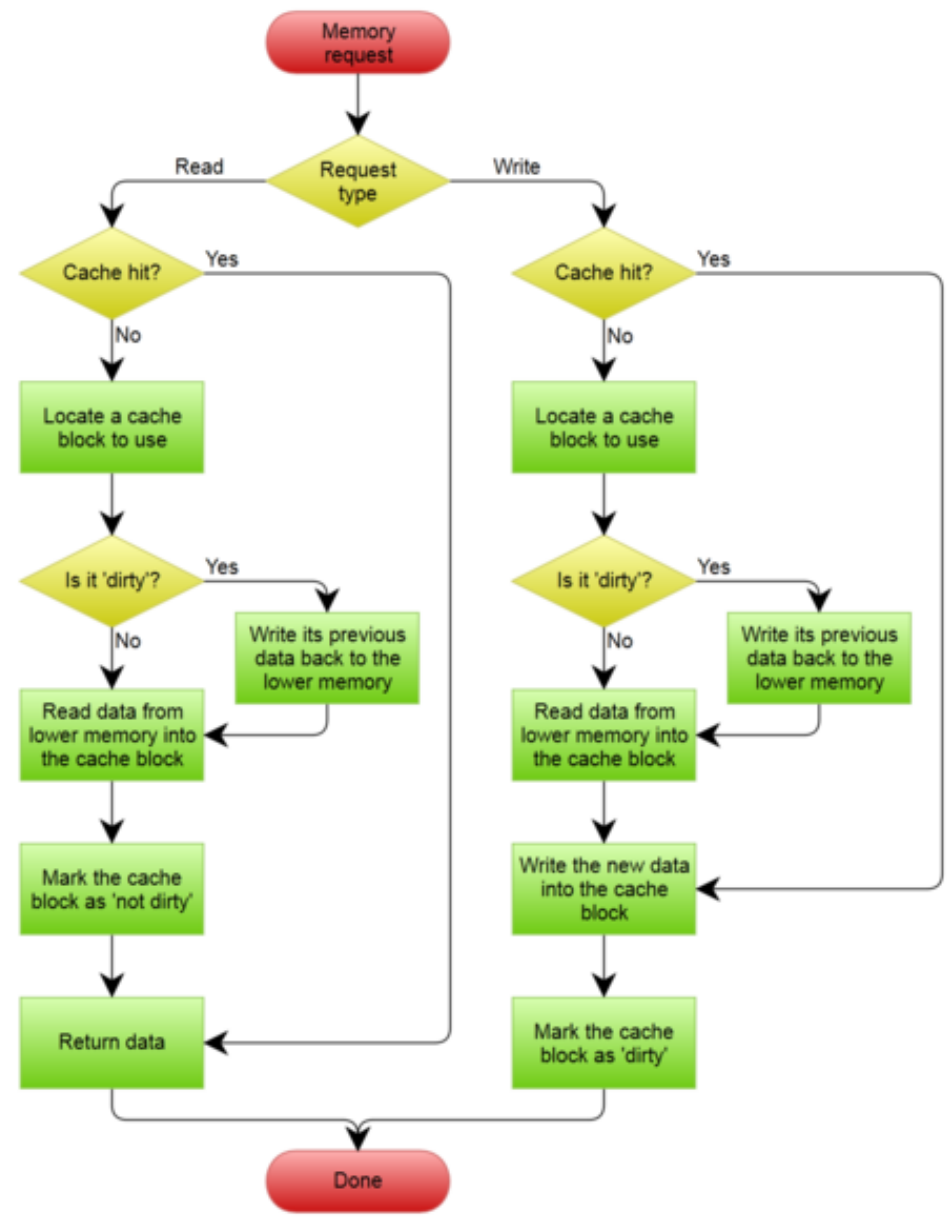
A Write-Back cache with Write Allocation
Read allocate

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?

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):

		no-write allocate	write allocate
1	write Mem[100];	miss	miss
2	write Mem[100];	miss	hit
3	read Mem[200];	miss	miss
4	write Mem[200];	hit	hit
5	write Mem[100];	miss	hit

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

Summary

Read Cache

Read through

Read allocate

Write Cache

hit

miss

Write-through

write is done synchronously both to the cache and to the backing store

Write allocate

data at the missed-write location is loaded to cache, followed by a write-hit operation.

Write-back

writing is done only to the cache

No-write allocate

data at the missed-write location is not loaded to cache, and is written directly to the backing store

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

$$\text{Average Memory Access Time} = \frac{\text{Whole accesses time}}{\text{All memory accesses in program}}$$

$$= \frac{\text{Accesses time on hitting+ Accesses time on miss}}{\text{All memory accesses in program}}$$

$$= \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

$$= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst} \right) \times Inst\%$$

$$\left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data} \right) \times Data\%$$

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

Example: Impact on Performance

- Assume :
    - *all instructions usually take 1.0 clock cycles (ignoring memory stalls).*
    - *cache miss penalty is 200 clock cycles*
    - *the average miss rate is 2%*
    - *there is an average of 1.5 memory references per instruction*
    - *the average number of cache misses per 1000 instructions is 30*

With a cache   CPI =7
Without a cache   CPI = 1.0+200 ×1.5=301

- What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

Answer: CPU $_{time}$= (CPU clock cycles + IC × memory stall cycles/Inst)×CycleTime

= IC ×(CPI + memory stall cycles/Inst) ×CycleTime

= IC ×[1.0+(30/1000×200)] ×clock cycle =IC ×7.00 ×clock cycle

- Now calculating performance using miss rate, first compute memory stall cycles：

$$CPU\ time = IC \times \left(CPI_{execution} + Miss\ rate \times \frac{Memory\ accesses}{Instruction} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

$$CPU\ time_{with\ cache} = IC \times [1.0 + (1.5 \times 2\% \times 200)] \times Clock\ cycle\ time$$

$$= IC \times 7.00 \times Clock\ cycle\ time$$

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 ≠ 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
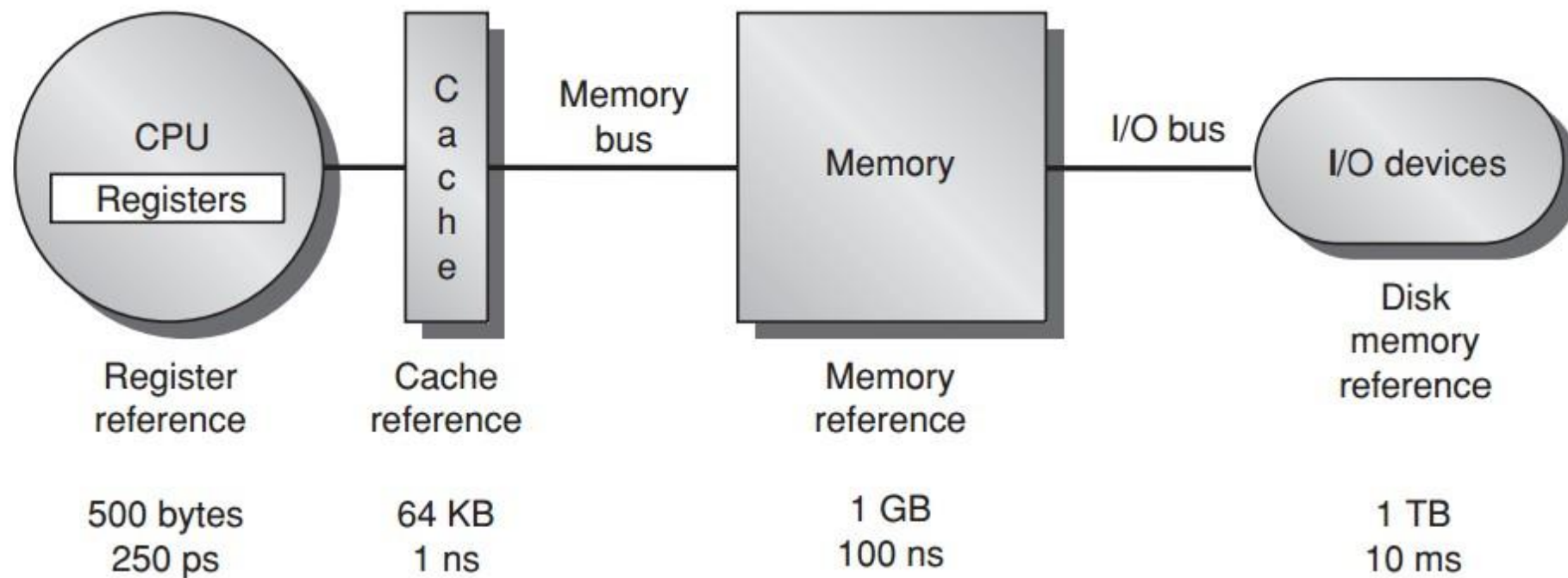
CPU Registers	Cache	Memory	I/O devices
Register reference	Cache reference	Memory reference	Disk memory reference
500 bytes 250 ps	64 KB 1 ns	1 GB 100 ns	1 TB 10 ms

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).

```
        Proc
     Unified
     Cache-1
  Unified
  Cache-2
```

```
  I-Cache-1      Proc      D-Cache-1
            Unified
            Cache-2
```

Memory Hierarchy

CPU Registers	Cache	Memory	I/O devices Disk
Register reference	Cache reference	Memory reference	Disk memory reference
500 bytes 250 ps	64 KB 1 ns	1 GB 100 ns	1 TB 10 ms

Memory Hierarchy

addr

CPU
Registers

Cache

Memory

I/O bus

I/O devices

Memory
bus

Register
reference

Cache
reference

Memory
reference

Disk
memory
reference

500 bytes
250 ps

64 KB
1 ns

1 GB
100 ns

1 TB
10 ms
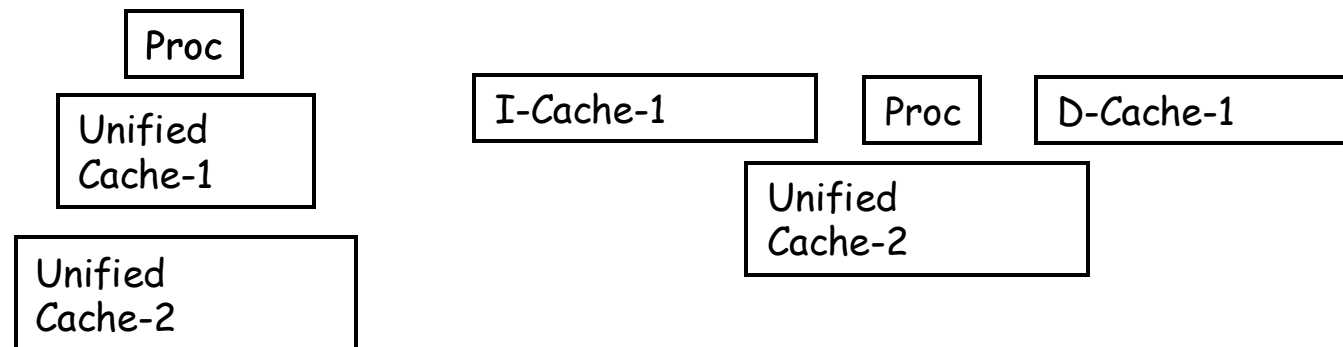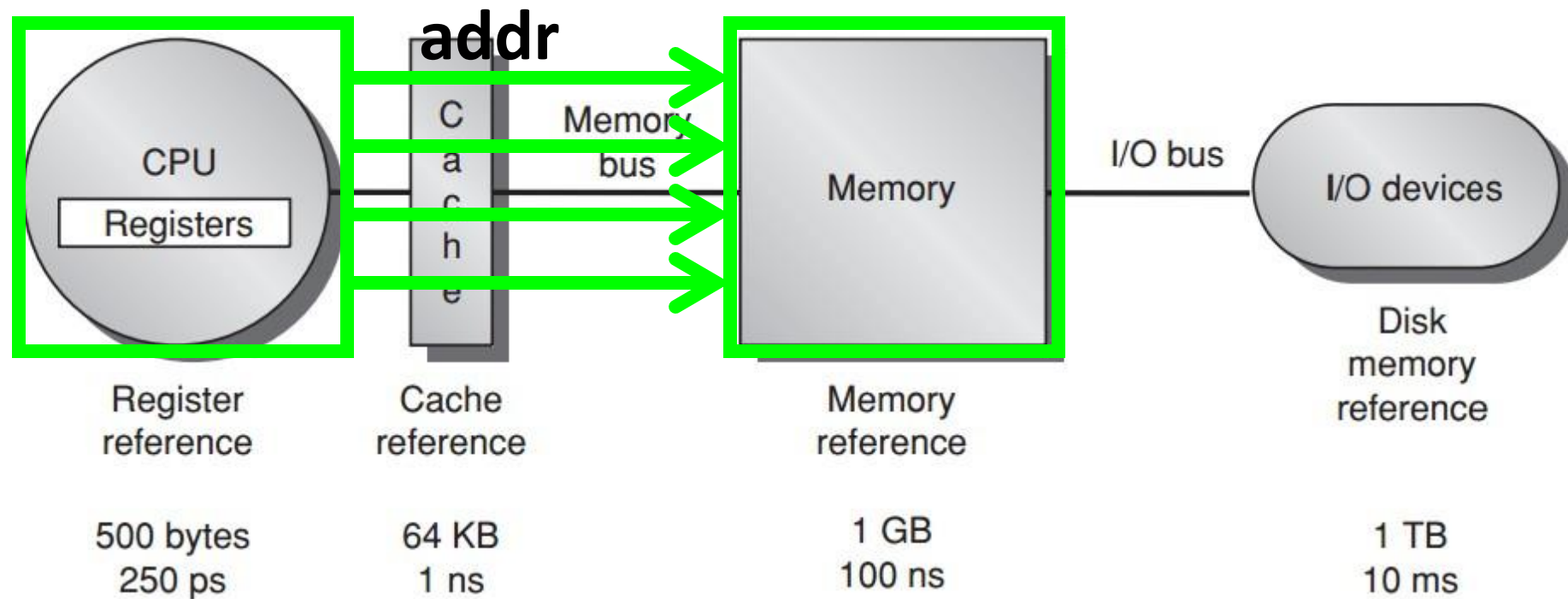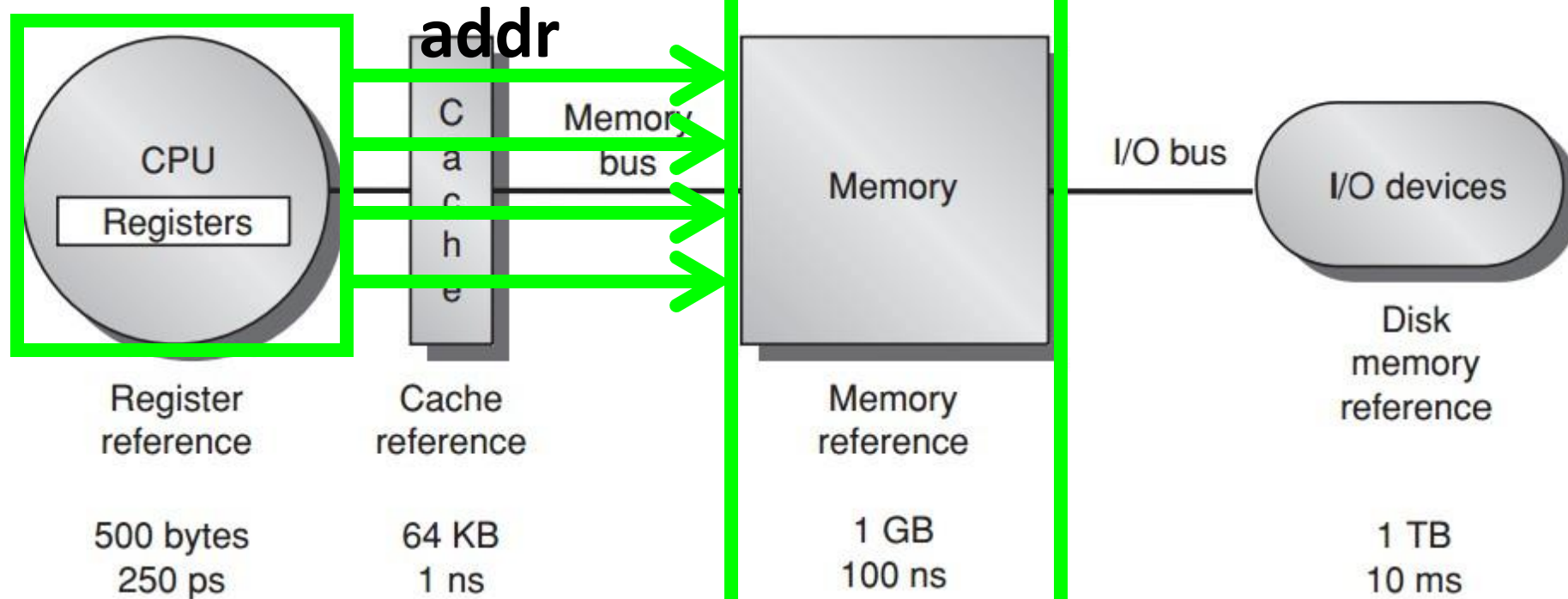
Larger memory for more processes?

Virtual Memory

Virtual Memory

Virtual
address

0	A
4K	B
8K	C
12K	D

Virtual memory

Physical
address

0	
4K	C
8K	
12K	
16K	A
20K	
24K	B
28K	

Physical
main memory

Disk

D
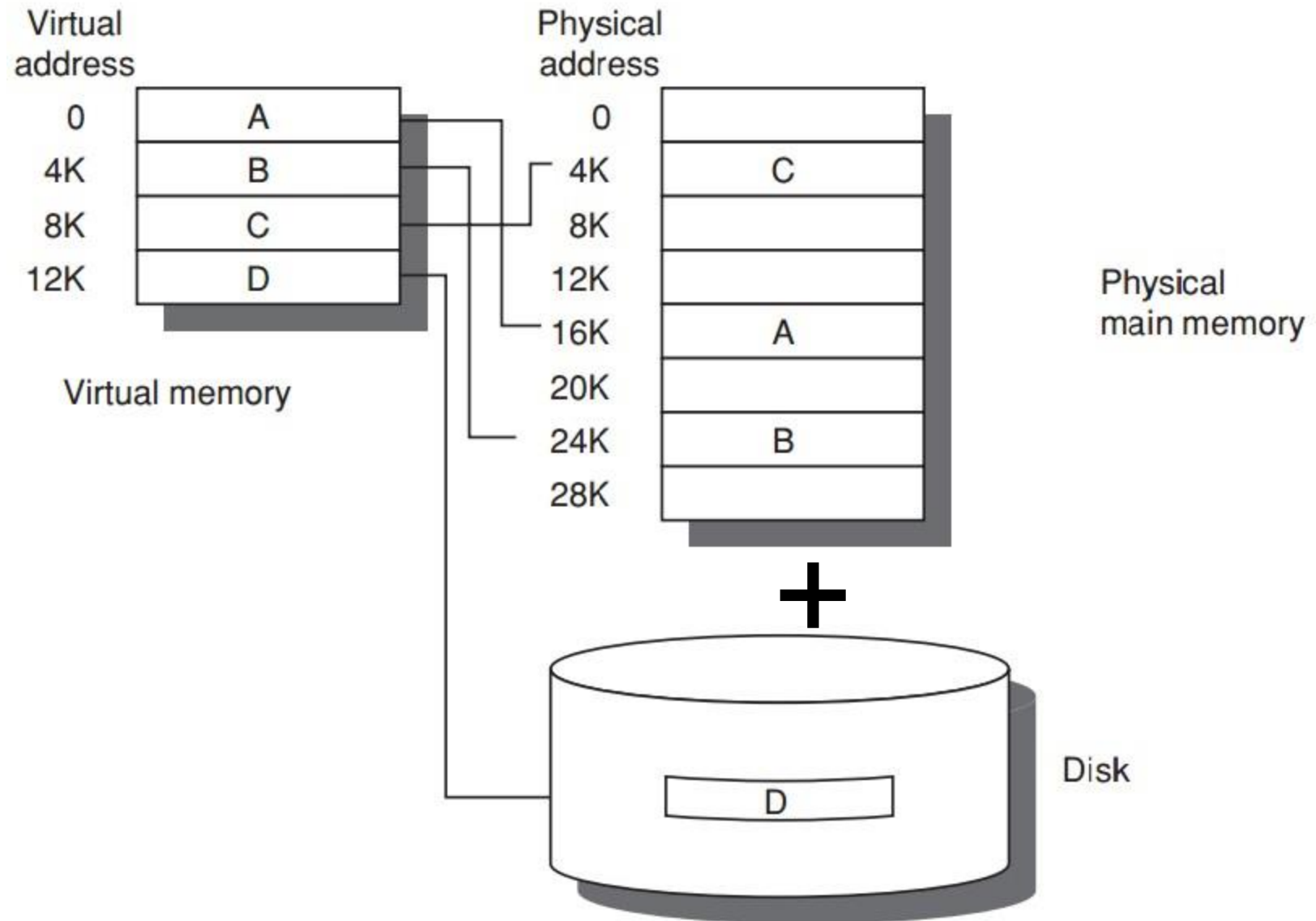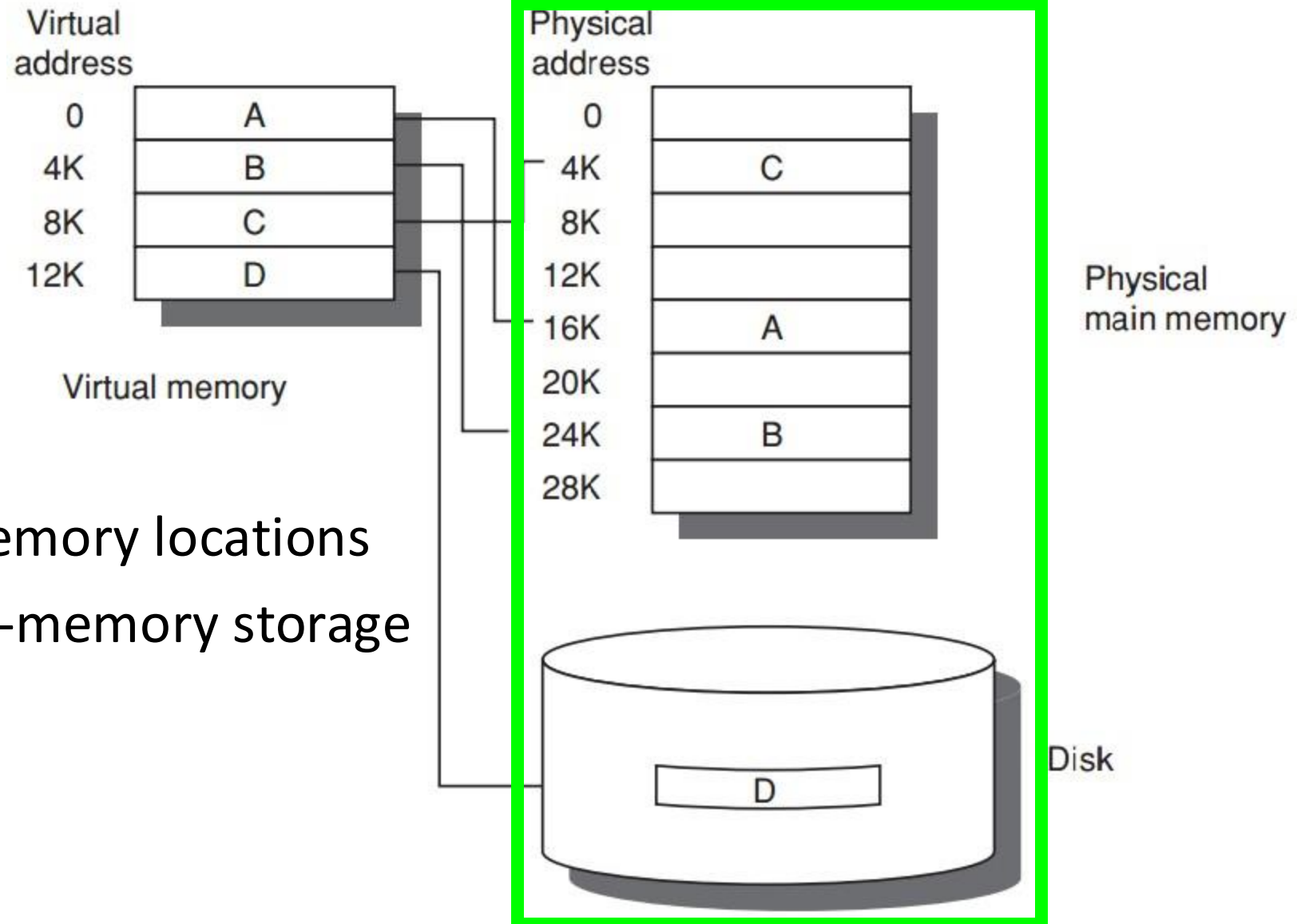
Program **uses**

• discontiguous memory locations

• + secondary/non-memory storage

Virtual Memory

Virtual
address

0	A
4K	B
8K	C
12K	D

Virtual memory

Physical
address

0	
4K	C
8K	
12K	
16K	A
20K	
24K	B
28K	

Physical
main memory

Disk

| D |
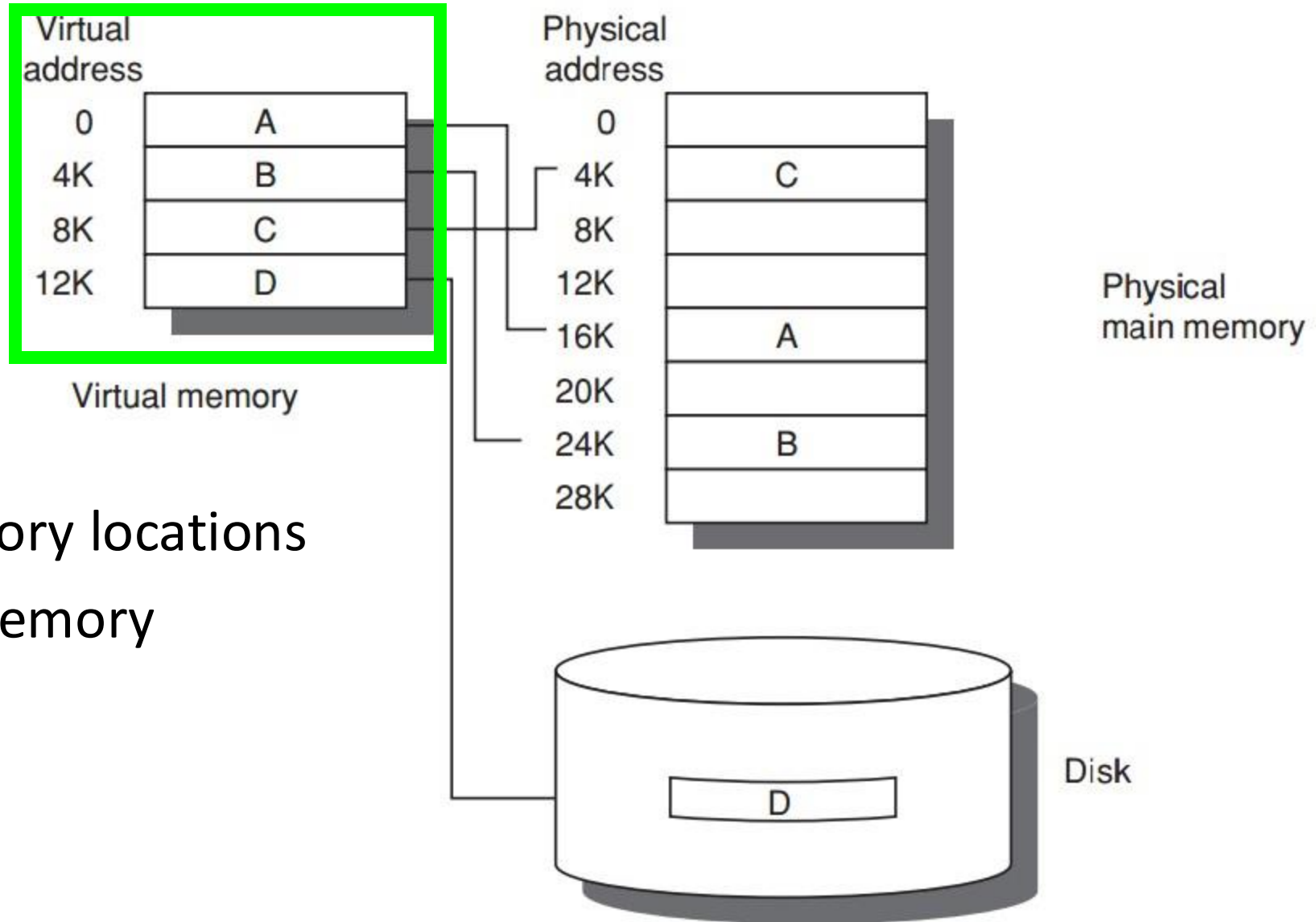
Program **thinks**

• 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

Main/Physical Memory

Process A

allocated

Process B
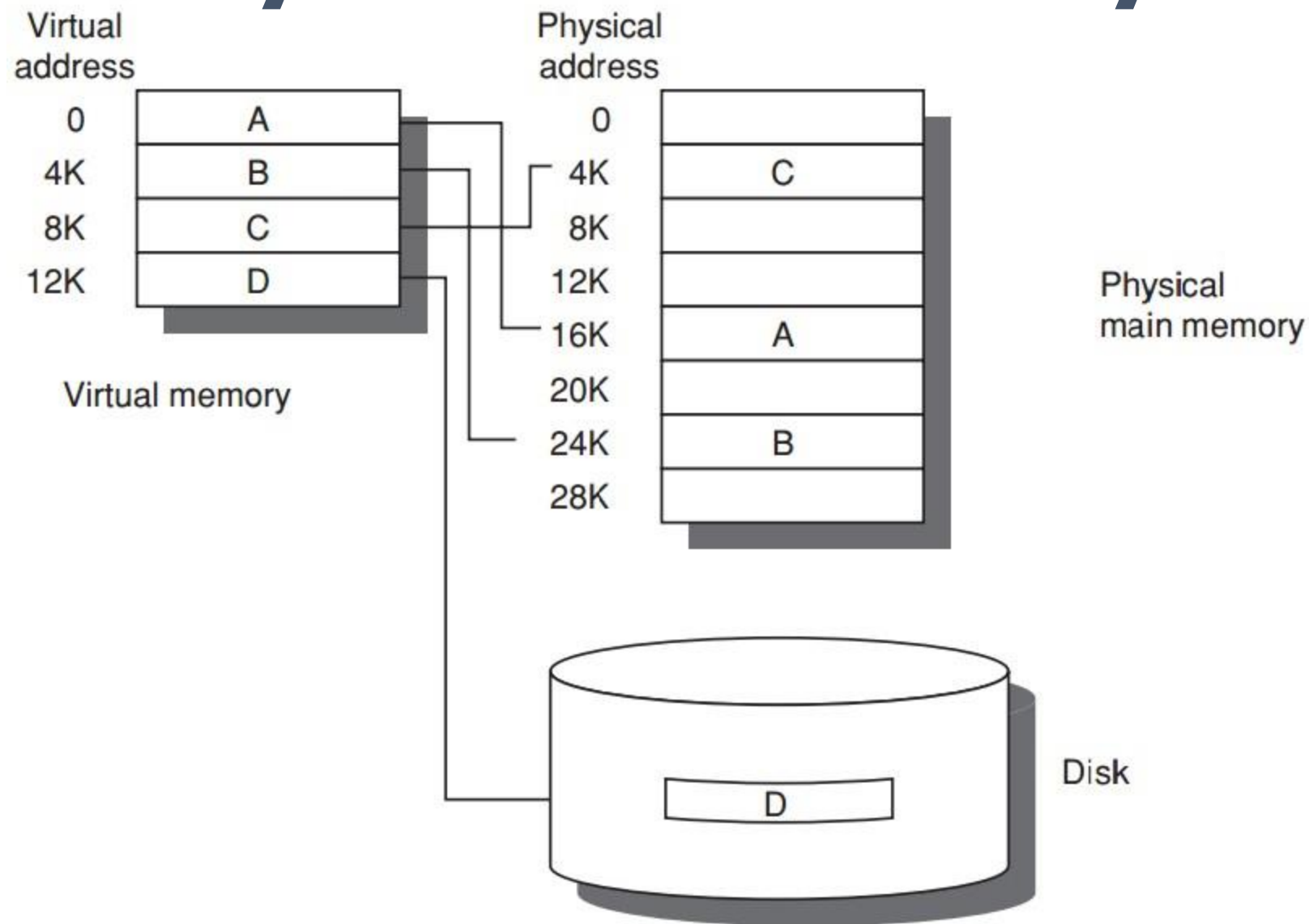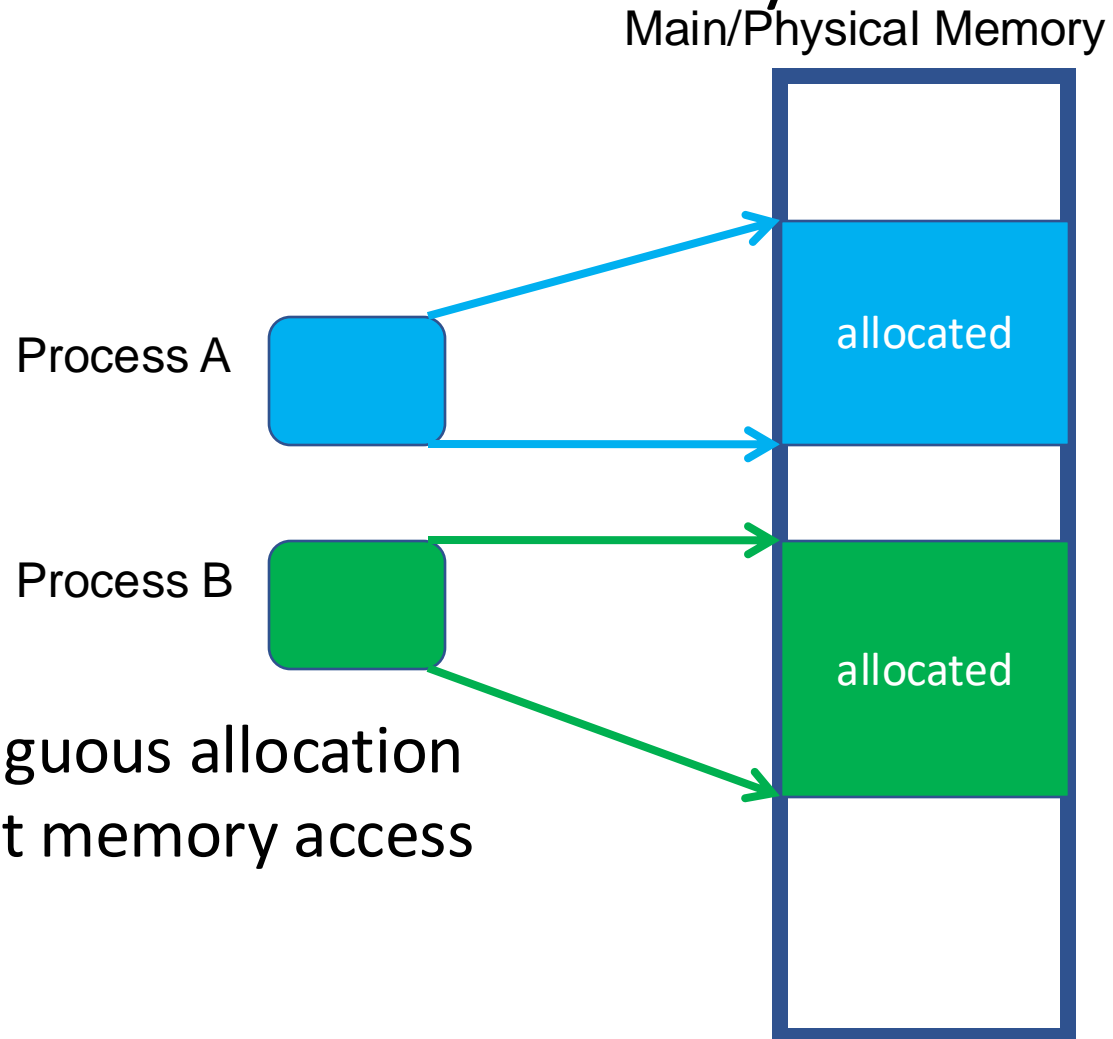
allocated

- Contiguous allocation
- Direct memory access

Prior Virtual Memory

Main/Physical Memory
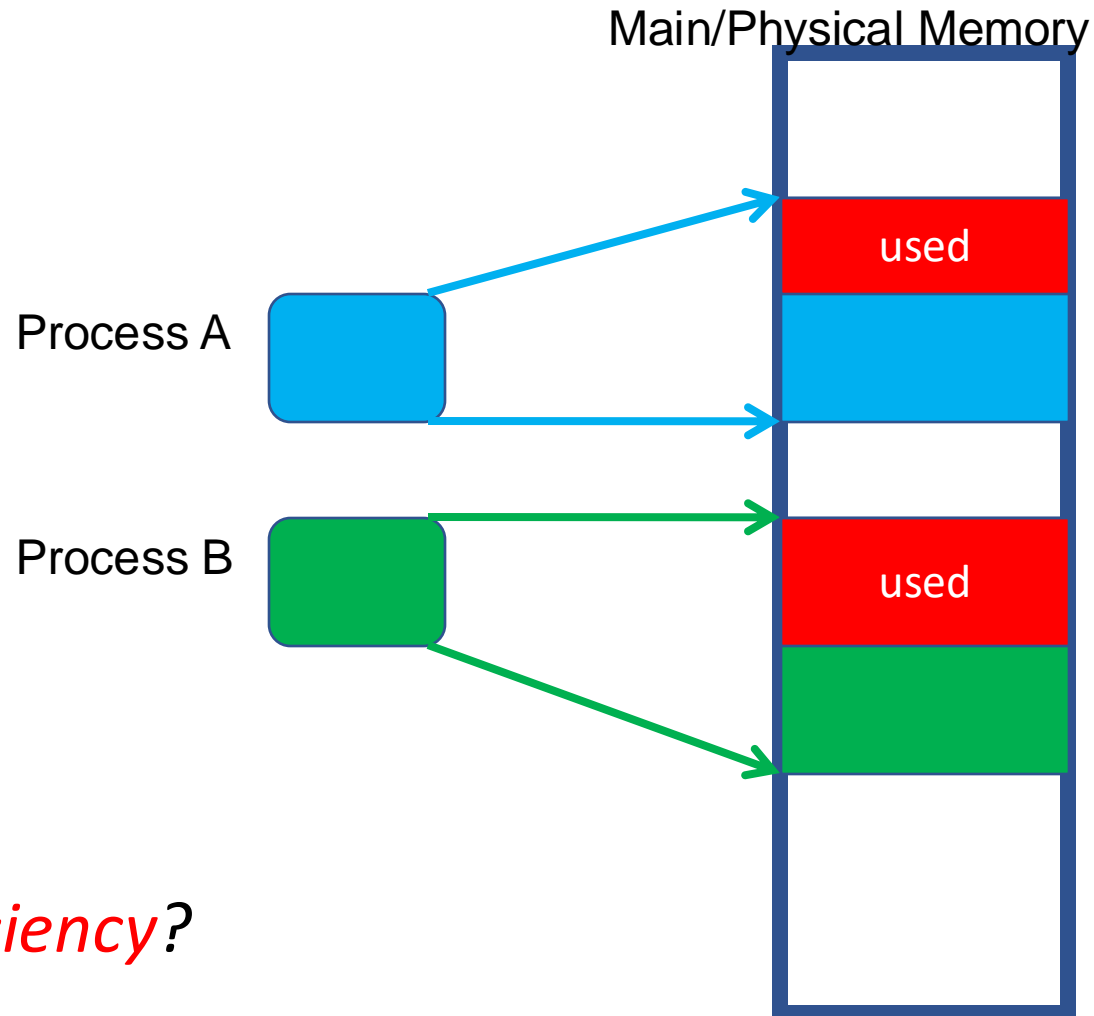
- *efficiency?*

Prior Virtual Memory

Main/Physical Memory

Process A

Process B

used

used
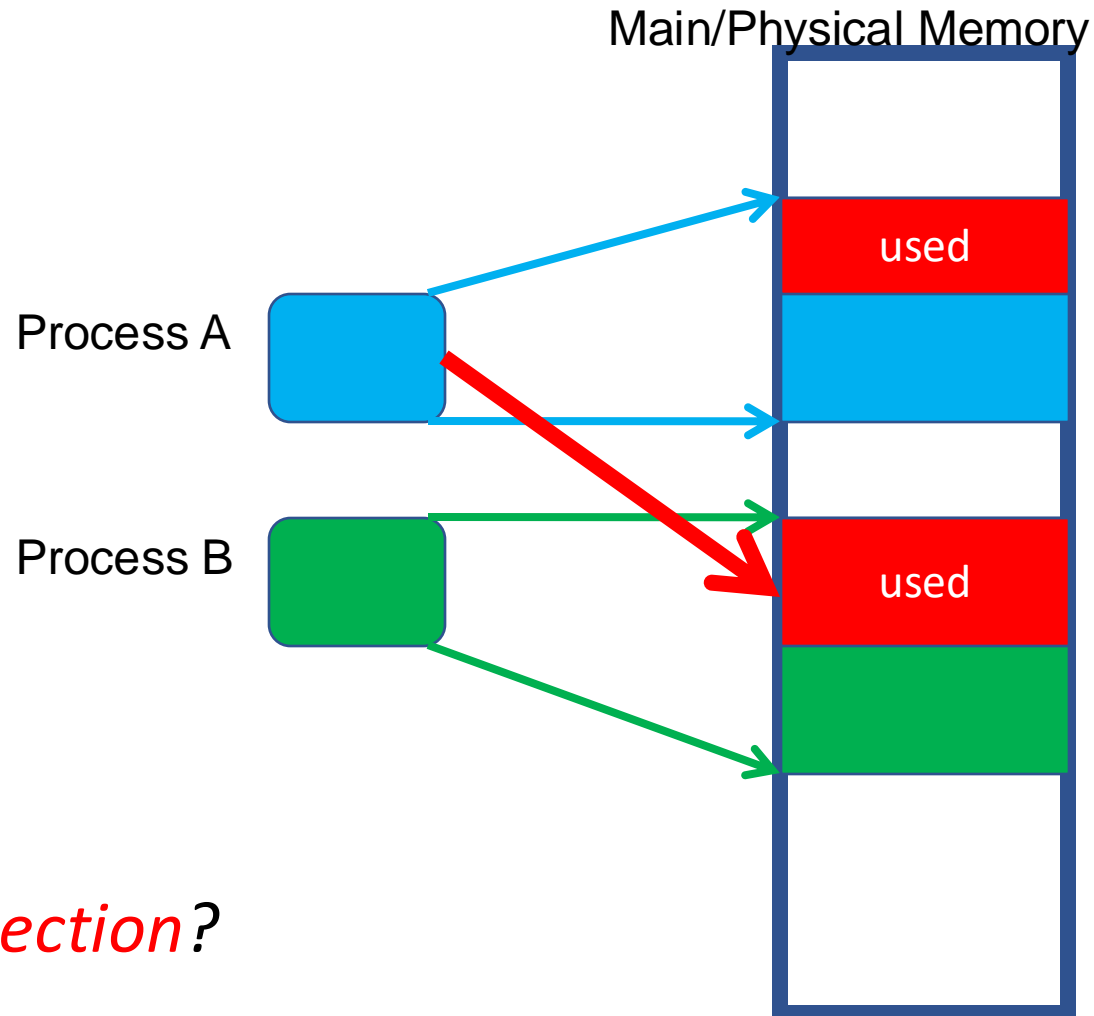
- *protection?*

Virtual Memory

Main/Physical Memory

Virtual memory

used

Process A

used

used

Process B

used
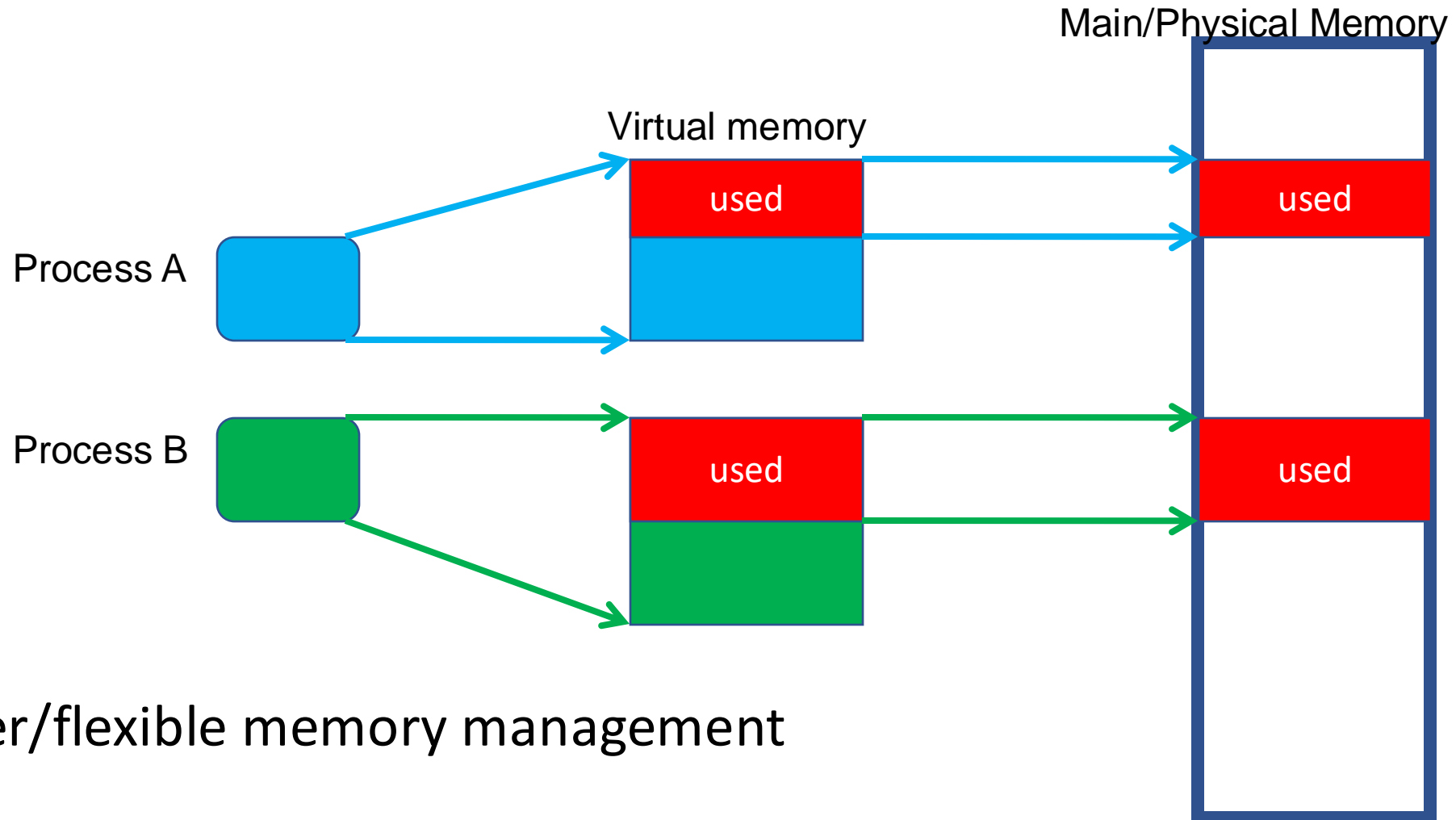
- Easier/flexible memory management

Virtual Memory
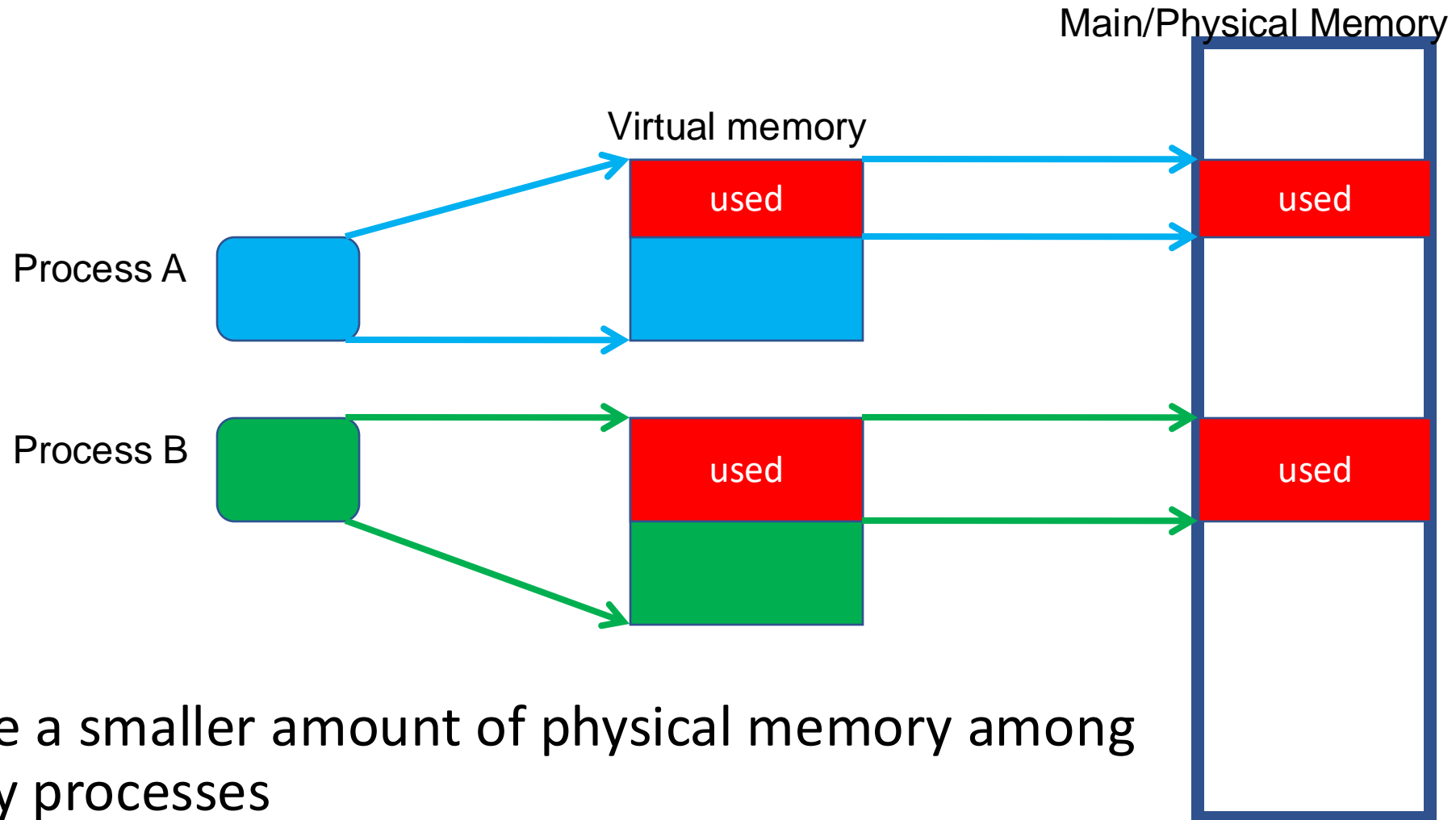
- Share a smaller amount of physical memory among many processes

Virtual Memory
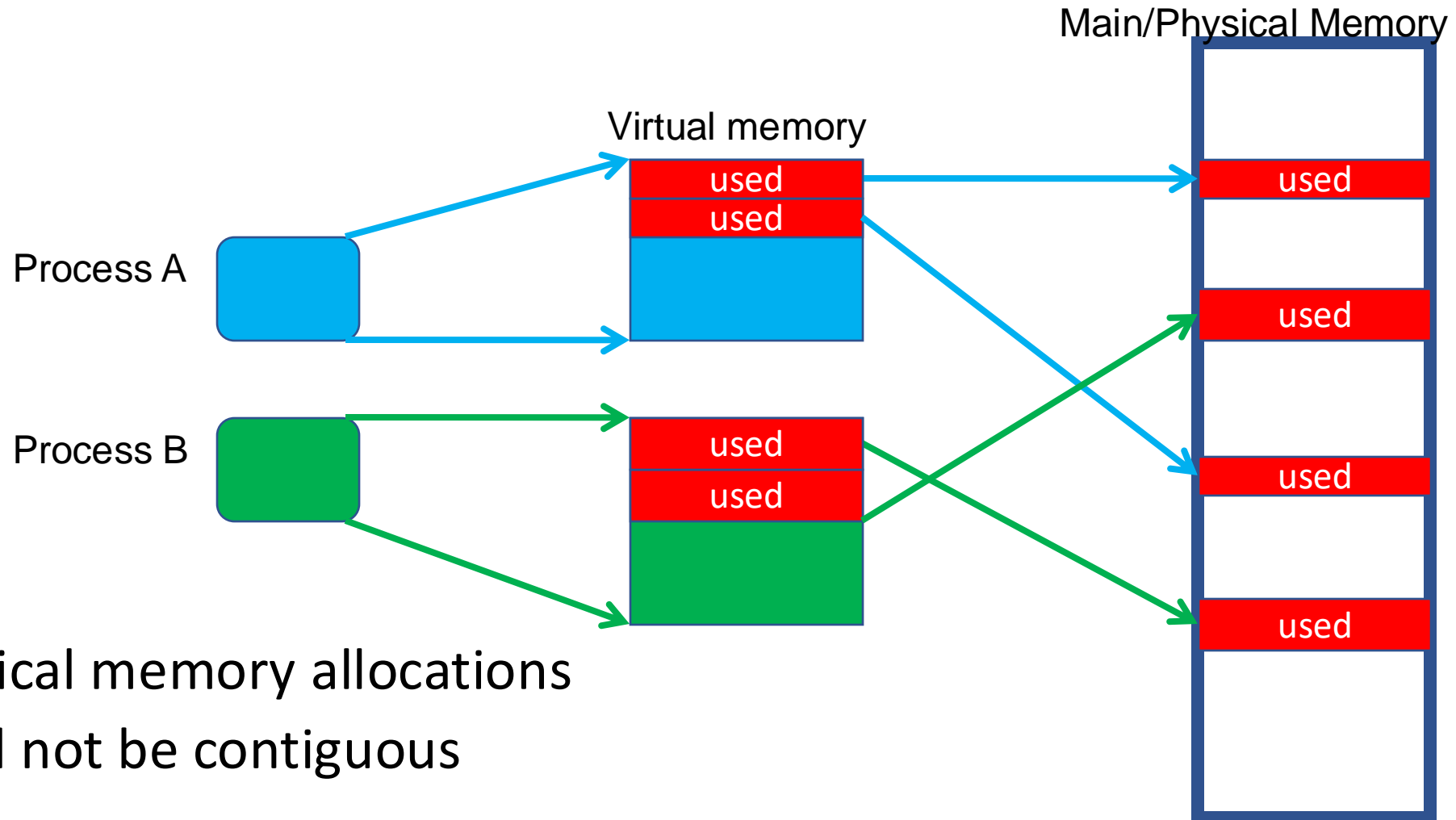
- Physical memory allocations need not be contiguous

Virtual Memory

Main/Physical Memory

Virtual memory

Process A

Process B

used
used

used
used

used

used

used

used

- memory protection; process isolation

Virtual Memory

Main/Physical Memory

Virtual memory

Process A

used
used

Process B

used
used

used

used

used

used

- Introduces another level of secondary storage

Virtual Memory =
Main Memory + Secondary Storage

CPU Registers	Cache	Memory	I/O devices
Register reference	Cache reference	Memory reference	Disk memory reference
Size: 500 bytes	64 KB	1 GB	1 TB
Speed: 250 ps	1 ns	100 ns	10 ms

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

	Code	Data
Paging		
Segmentation		
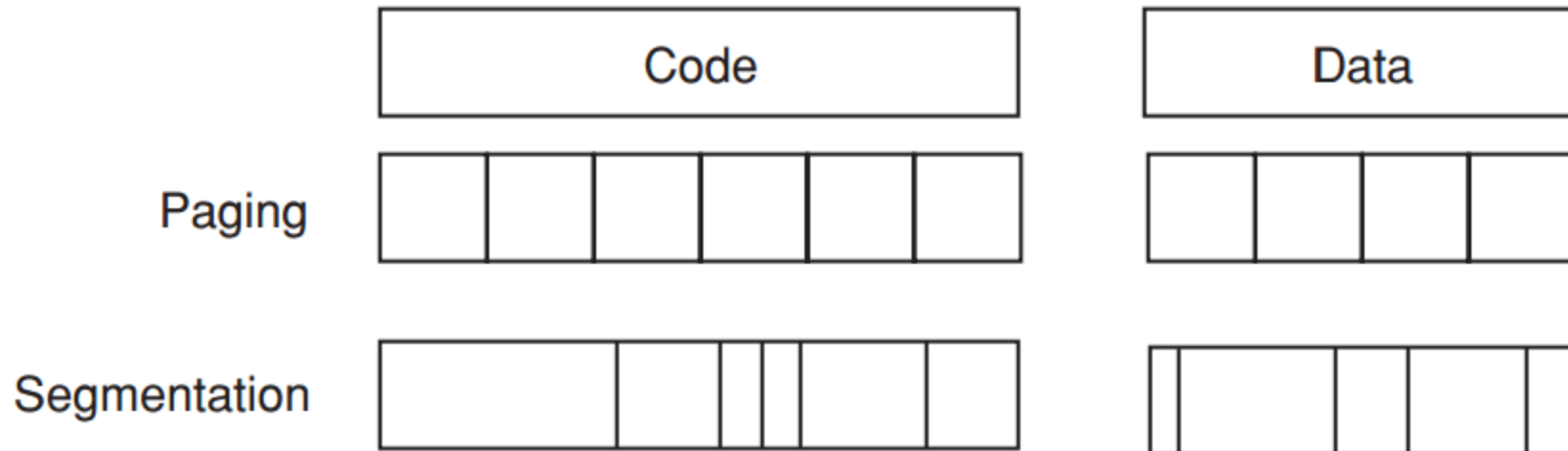
Virtual Memory

Address

- Paged virtual memory

page address: page # || offset

one word

concatenate

- Segmented virtual memory

segment address: seg # offset

two words
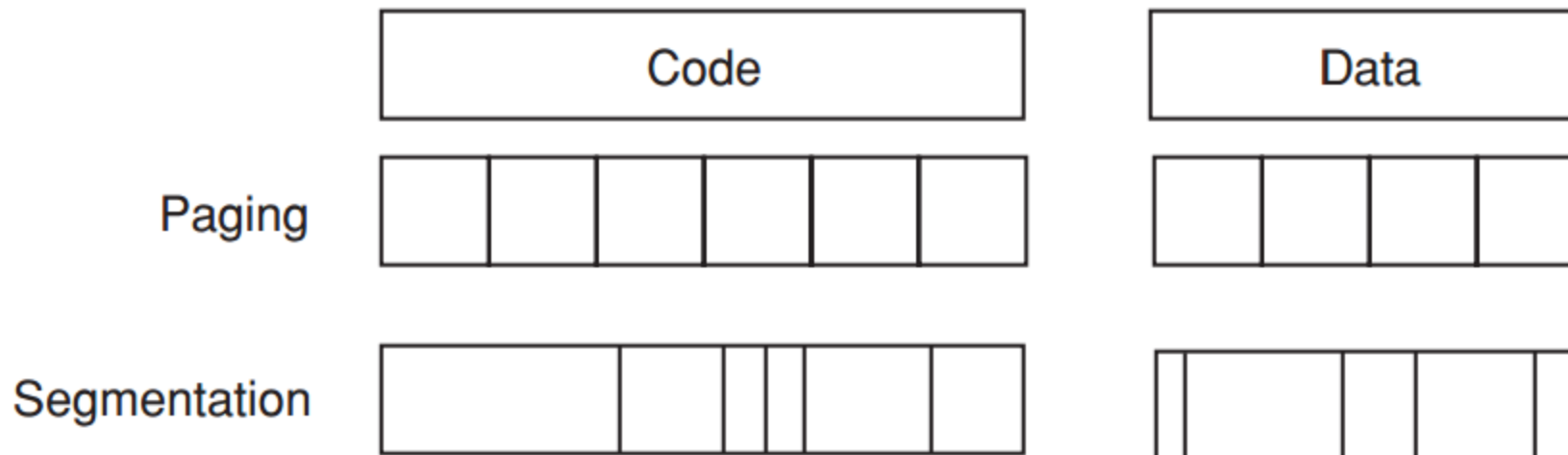
Pros & Cons?
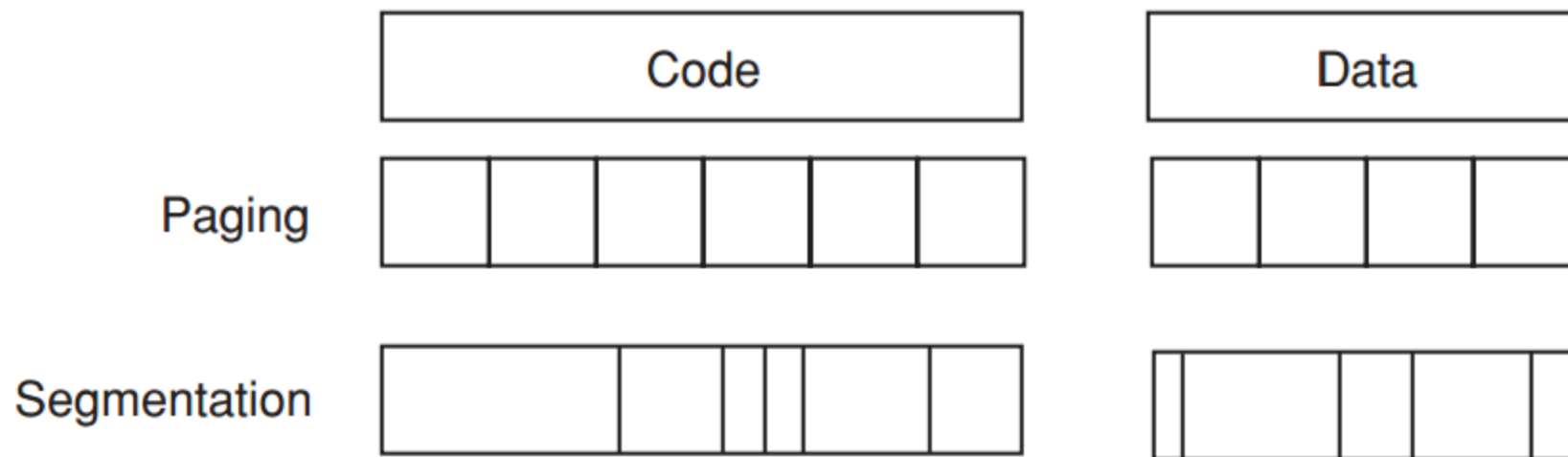
Paging vs Segmentation

	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 contiguous, variable-size, unused portion of main memory)
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)

http://www.cnblogs.com/felixfang/p/3420462.html

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?***
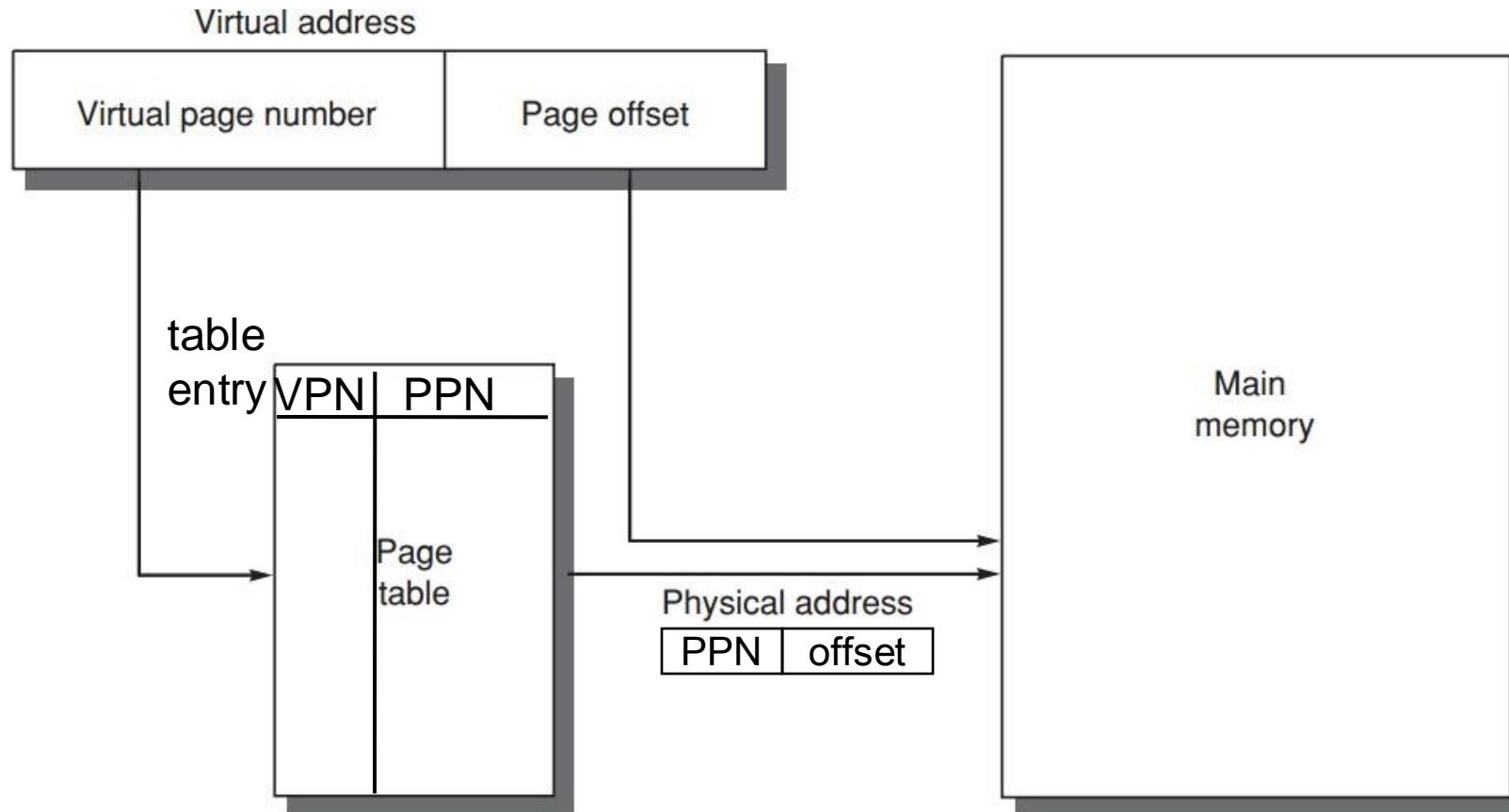
Four Memory Hierarchy Q's

- **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

How virtual memory works?

Virtual Memory

- Introduces another level of secondary storage
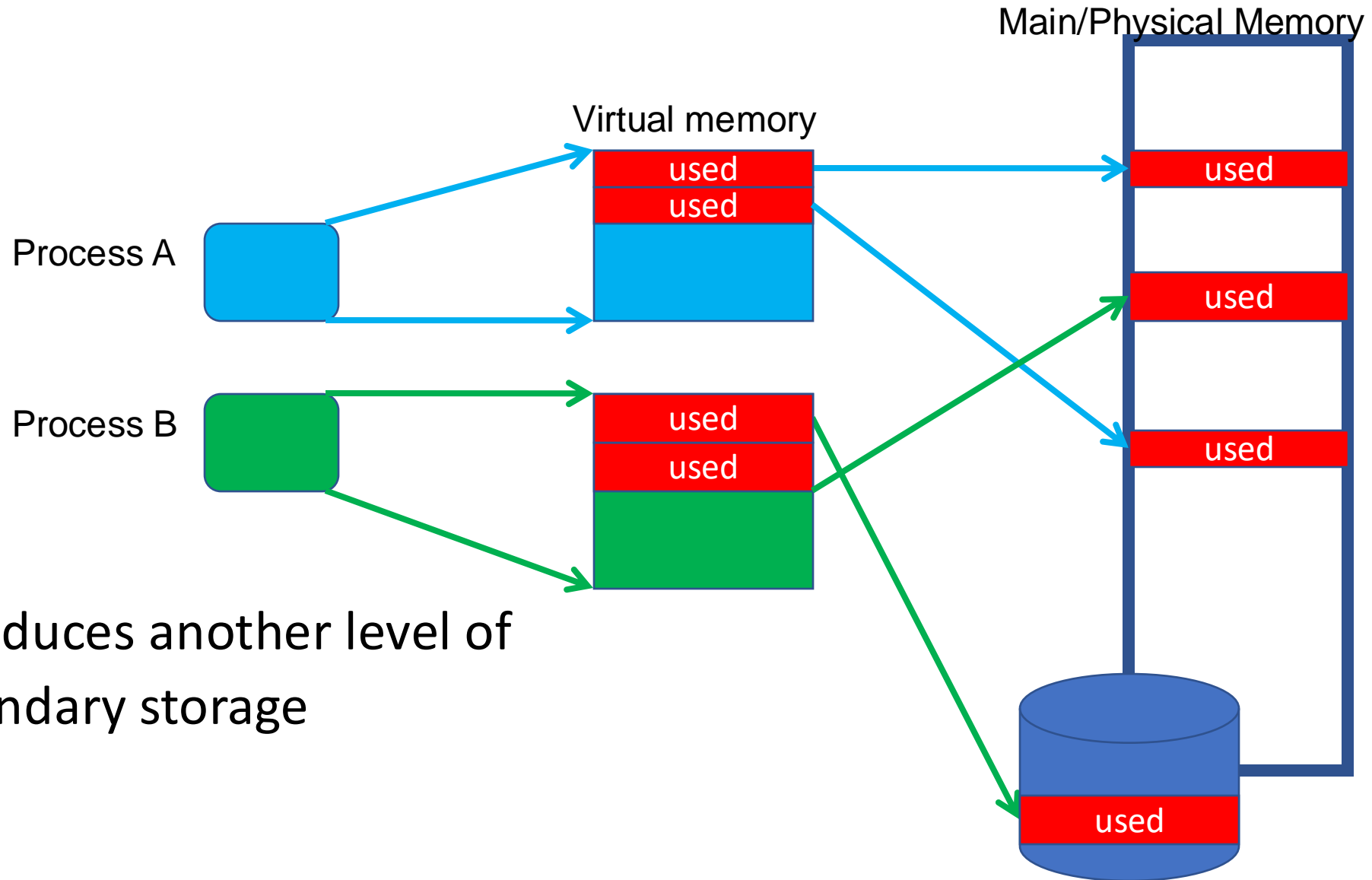
well...tell me more
Address Translation

Page Table

physical page number || page offset = physical address

Virtual address

| Virtual page number | Page offset |

page table

Main memory

Physical address
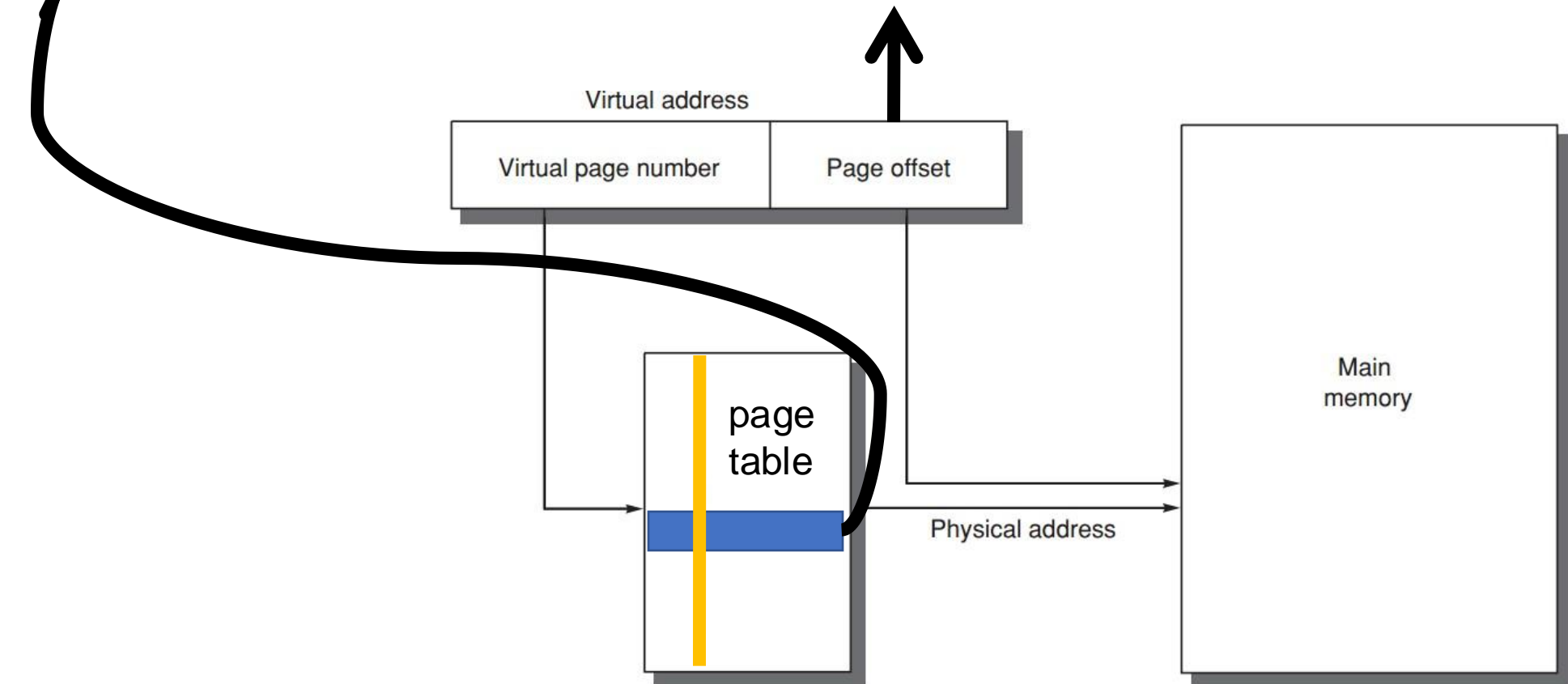
Page Table ?

- Page tables are often large

  32-bit virtual address, 4KB pages,

  4 bytes per page table entry.

  page table size:

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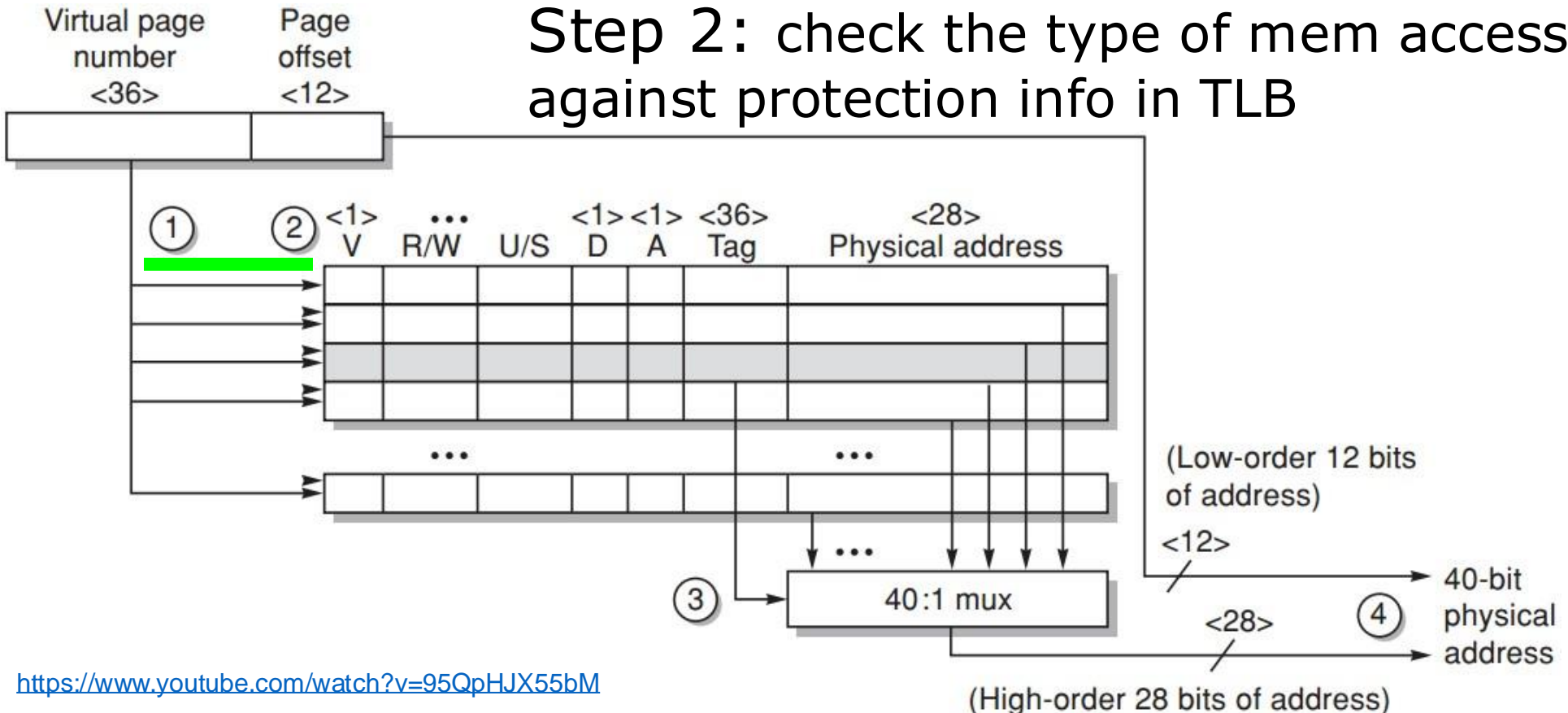
TLB Example

- **Opteron data TLB**

  Steps 3: the matching tag sends phy addr through multiplexor
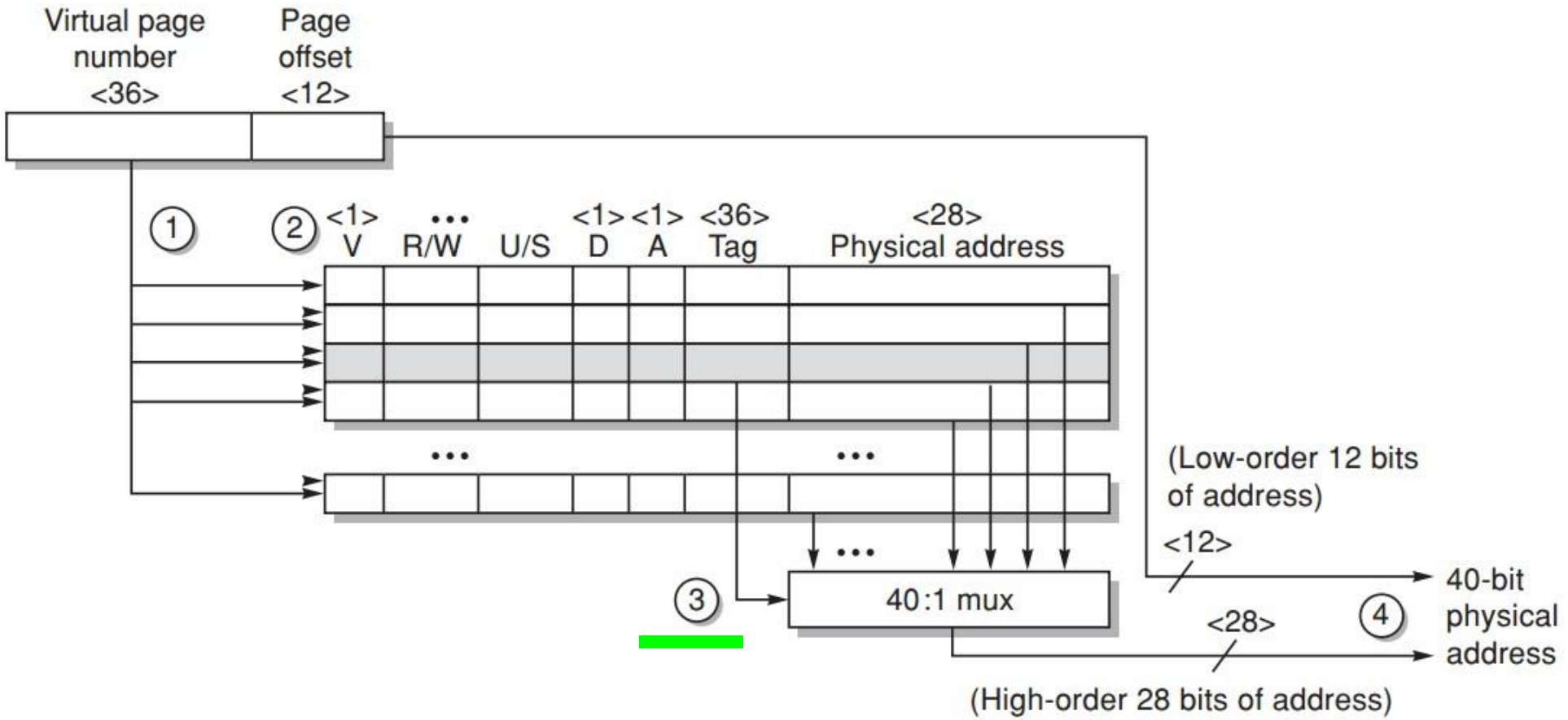
TLB Example

- **Opteron data TLB**

Steps 4: concatenate page offset to phy page frame to form final phy addr
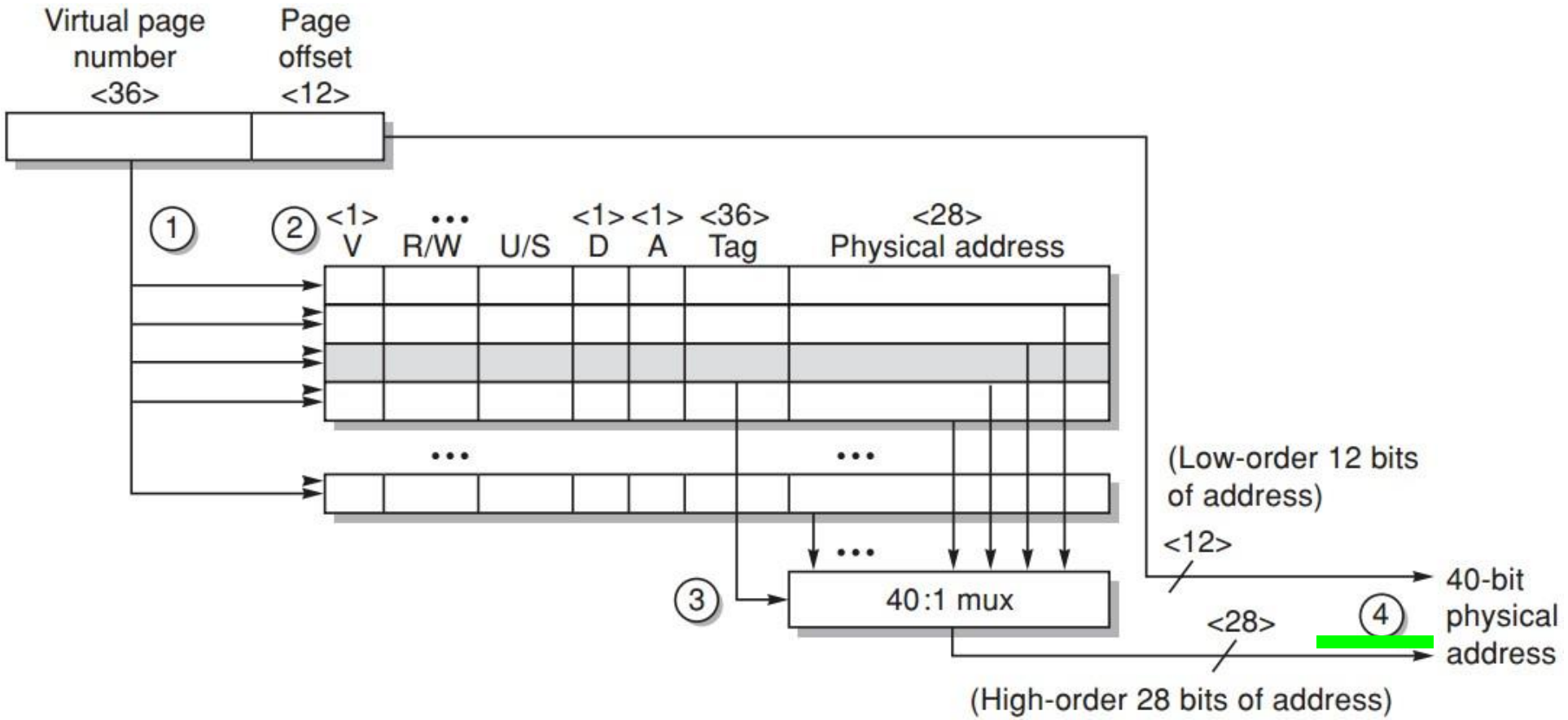
Does page size matter?

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
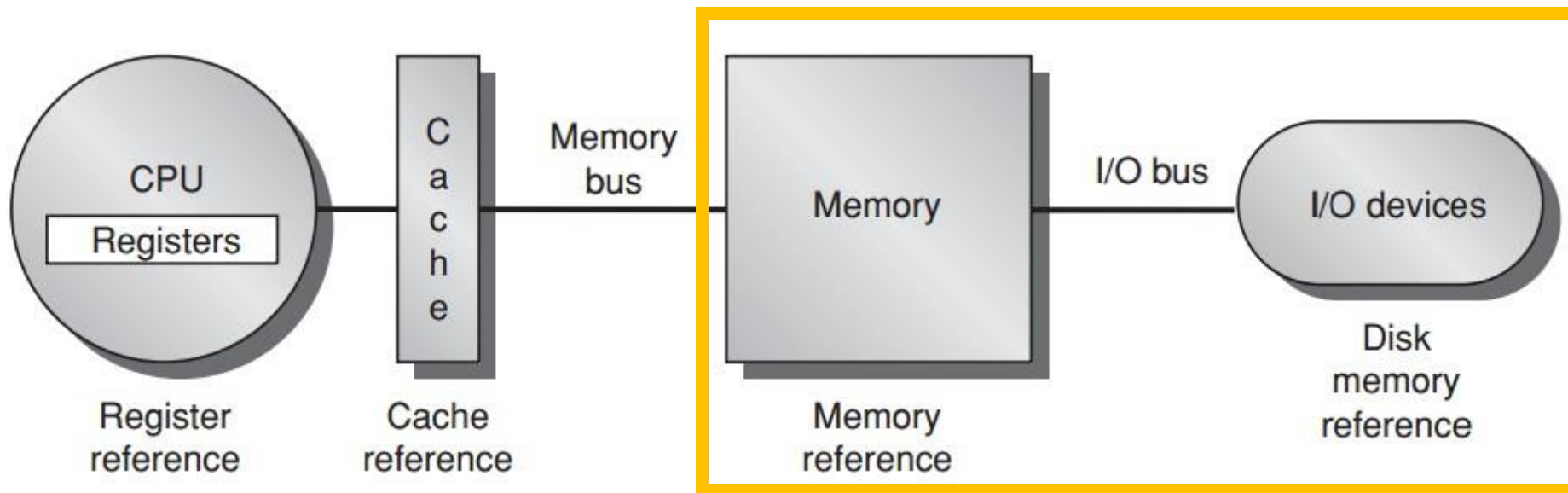
Virtual Memory =
Main Memory + Secondary Storage
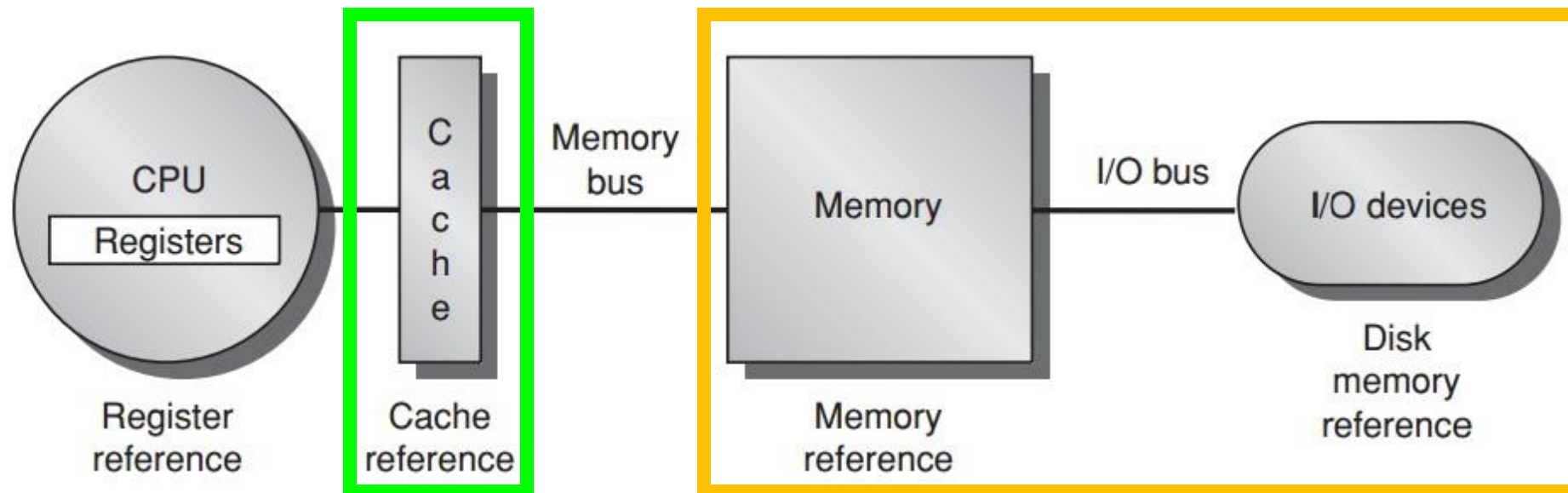
Address Translation:
one more time, with cache

Virtual address <64>

Virtual page number <51>   Page offset <13>

TLB tag compare address <43>   TLB index <8>   L1 cache index <7>   Block offset <6>

To CPU

TLB tag <43>   TLB data <28>   L1 cache tag <28>   L1 data <512>

L1 tag compare address <28>

To CPU

=?   =?

64-bit virtual address

41-bit physical address   Physical address <41>

page size: 8KB

two-level direct-mapped caches   L2 tag compare address <19>   L2 cache index <16>   Block offset <6>

64-byte blocks   To CPU

L1: 8KB

L2: 4MB   L2 cache tag <19>   L2 data <512>

TLB 256 entries

Address Translation

=?

To L1 cache or CPU

Virtual address <64>

Virtual page number <51> Page offset <13> 2^{13} = 8KB page

TLB tag compare address <43> TLB index <8>

L1 cache index <7> Block offset <6>

To CPU

TLB tag <43> TLB data <28>

L1 cache tag <28> L1 data <512>

L1 tag compare address <28>

To CPU

=?

=?

Physical address <41>

L2 tag compare address <19> L2 cache index <16> Block offset <6>

To CPU

L2 cache tag <19> L2 data <512>

=?

To L1 cache or CPU

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

Virtual address <64>

Virtual page number <51>   Page offset <13>

TLB tag compare address <43>   TLB index <8>   L1 cache index <7>   Block offset <6>

$2^8 = 256$ entries

TLB tag <43>   TLB data <28>

L1 cache tag <28>   L1 data <512>

To CPU

To CPU

L1 tag compare address <28>

=?   =?

64-bit virtual address

41-bit physical address

Physical address <41>

page size: 8KB

two-level direct-mapped caches

L2 tag compare address <19>   L2 cache index <16>   Block offset <6>

64-byte blocks

To CPU

L1: 8KB

L2: 4MB

L2 cache tag <19>   L2 data <512>

Address Translation

=?

To L1 cache or CPU

Virtual address <64>

Virtual page number <51>   Page offset <13>

TLB tag compare address <43>   TLB index <8>   L1 cache index <7>   Block offset <6>

$2^6 = 64$byte block

To CPU

28+ 13=41

TLB tag <43>   TLB data <28>   L1 cache tag <28>   L1 data <512>

L1 tag compare address <28>

To CPU

=?   =?

64-bit virtual address

41-bit physical address

page size: 8KB

Physical address <41>

two-level direct-mapped caches

L2 tag compare address <19>   L2 cache index <16>   Block offset <6>

64-byte blocks

To CPU

L1: 8KB

L2: 4MB

L2 cache tag <19>   L2 data <512>

TLB 256 entries

Address Translation

=?

To L1 cache or CPU

Virtual address <64>

Virtual page number <51>   Page offset <13>

TLB tag compare address <43>   TLB index <8>   L1 cache index <7>   Block offset <6>

2^6 = 64byte block

2^{13} = 8KB L1 cache
2^6 = 64byte block
$2^7 = 2^{13}/2^6$

To CPU

TLB tag <43>   TLB data <28>   L1 cache tag <28>   L1 data <512>

L1 tag compare address <28>

To CPU

=?   =?

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

Physical address <41>

L2 tag compare address <19>   L2 cache index <16>   Block offset <6>

To CPU

L2 cache tag <19>   L2 data <512>

Address Translation

=?

To L1 cache or CPU

Virtual address <64>

Virtual page number <51>   Page offset <13>

TLB tag compare address <43>   TLB index <8>   L1 cache index <7>   Block offset <6>

To CPU

TLB tag <43>   TLB data <28>   L1 cache tag <28>   L1 data <512>

L1 tag compare address <28>

To CPU

=?   =?

64-bit virtual address

41-bit physical address

page size: 8KB

two-level direct-mapped caches
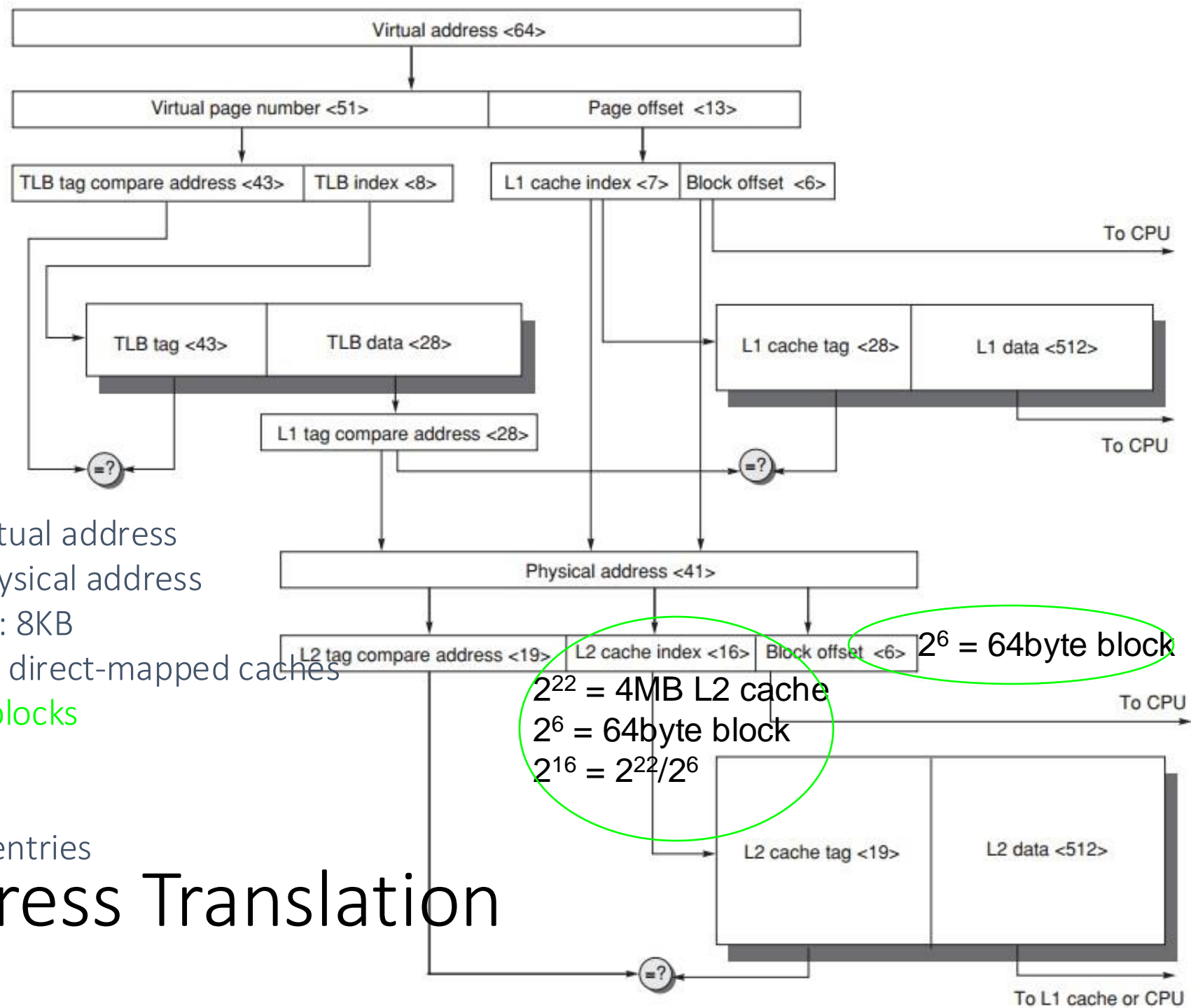
Physical address <41>

64-byte blocks

L1: 8KB

L2: 4MB

TLB 256 entries

L2 tag compare address <19>   L2 cache index <16>   Block offset <6> $2^6 = 64$byte block

$2^{22} = 4$MB L2 cache
$2^6 = 64$byte block
$2^{16} = 2^{22}/2^6$

To CPU

L2 cache tag <19>   L2 data <512>

Address Translation

To L1 cache or CPU

ready?

Virtual address <64>

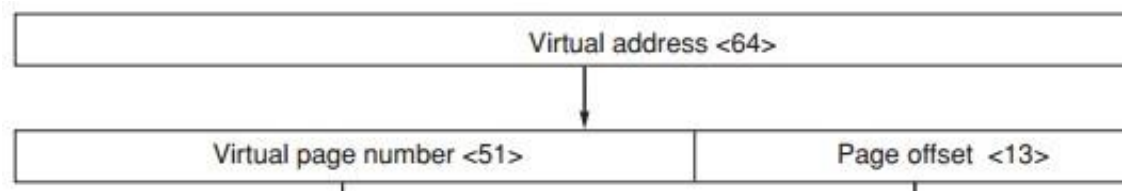| Virtual page number <51>	Page offset <13>

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

Virtual address <64>

Virtual page number <51> | Page offset <13>

TLB tag compare address <43> | TLB index <8> | L1 cache index <7> | Block offset <6>

To CPU

TLB tag <43> | TLB data <28> | L1 cache tag <28> | L1 data <512>

L1 tag compare address <28>

To CPU

=? =?

64-bit virtual address

41-bit physical address

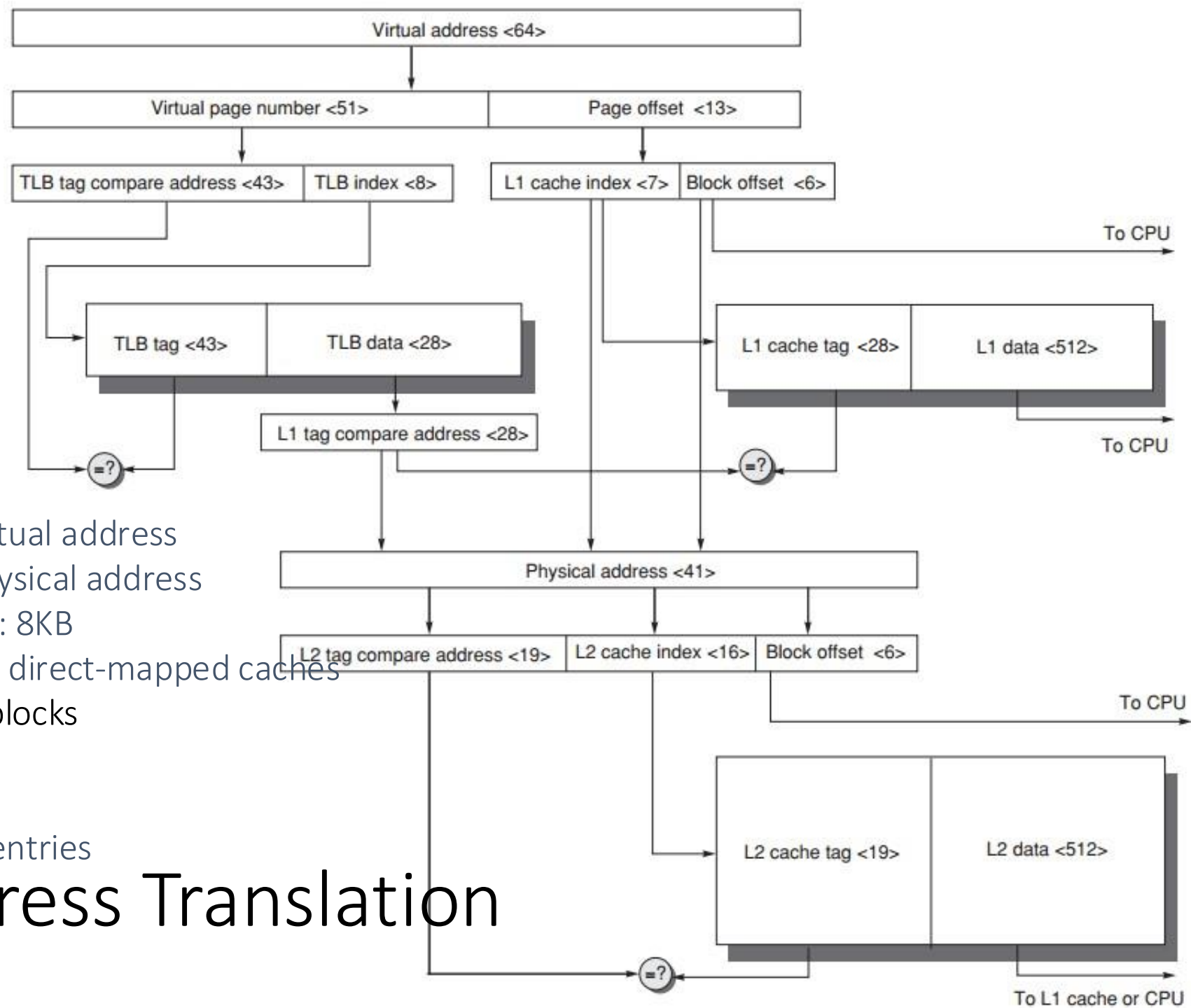page size: 8KB

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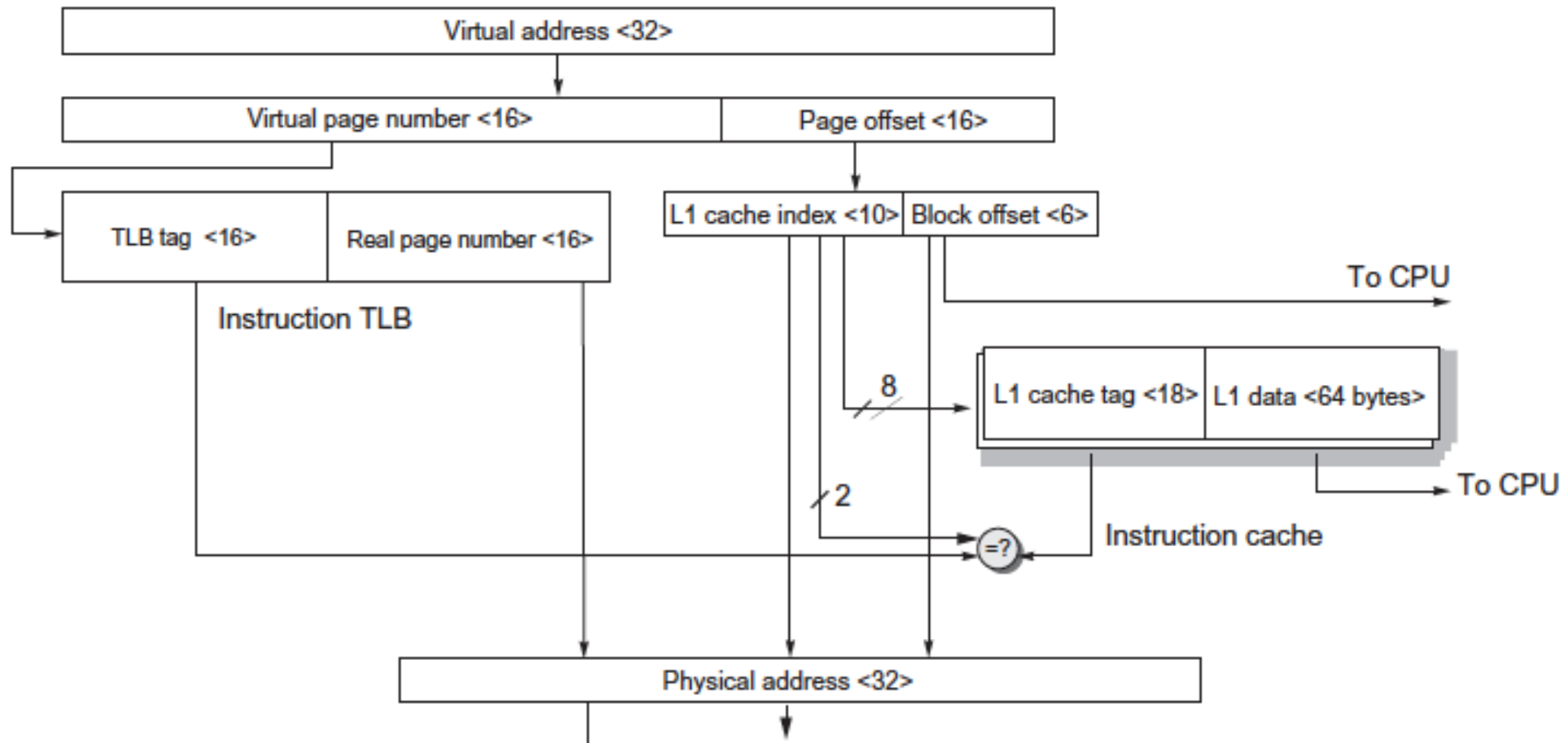page size: 8KB
two-level direct-mapped caches
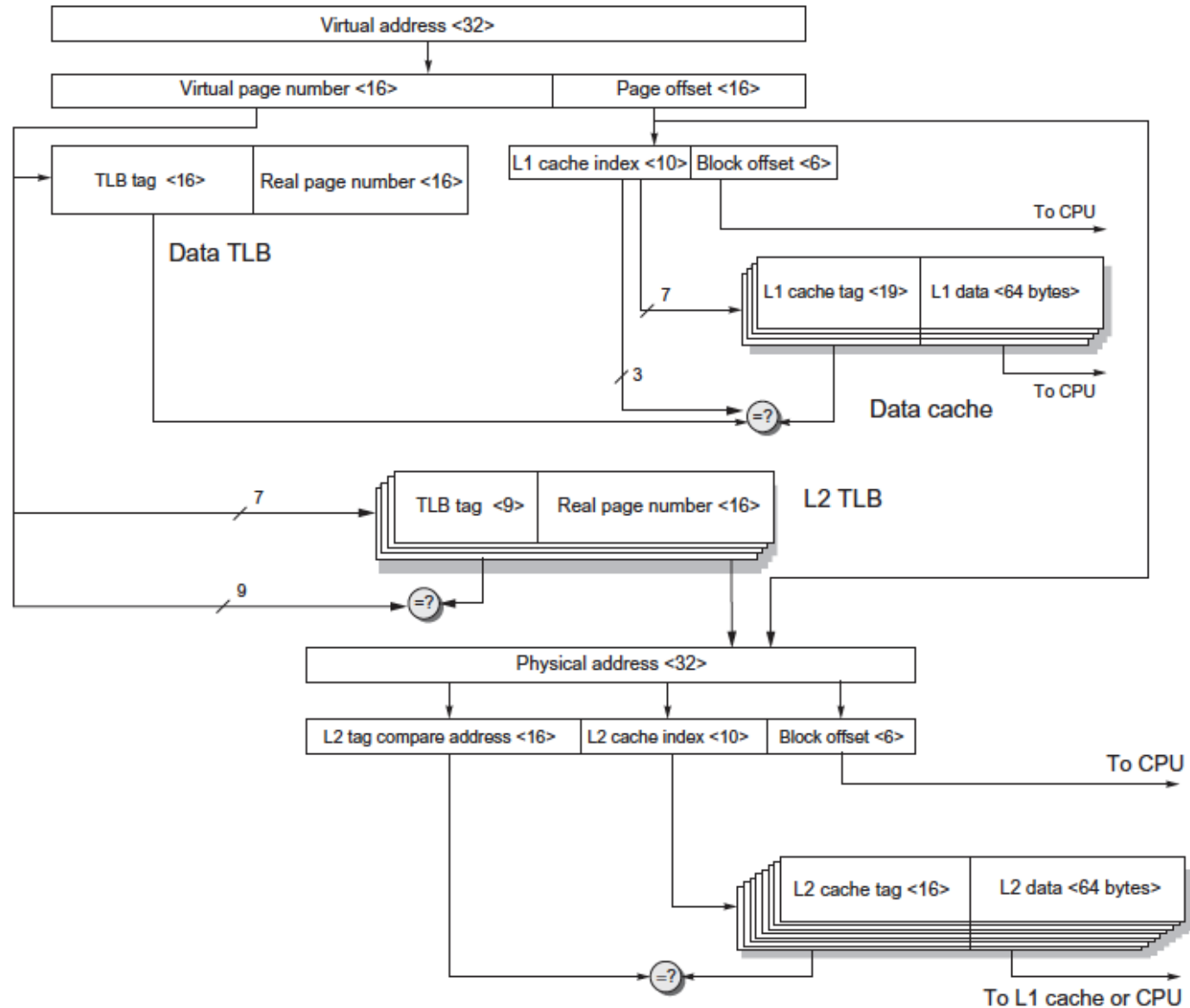64-byte blocks
L1: 8KB
L2: 4MB
TLB 256 entries

Address Translation

Memory Hierarchies in the ARM Cortex-A53
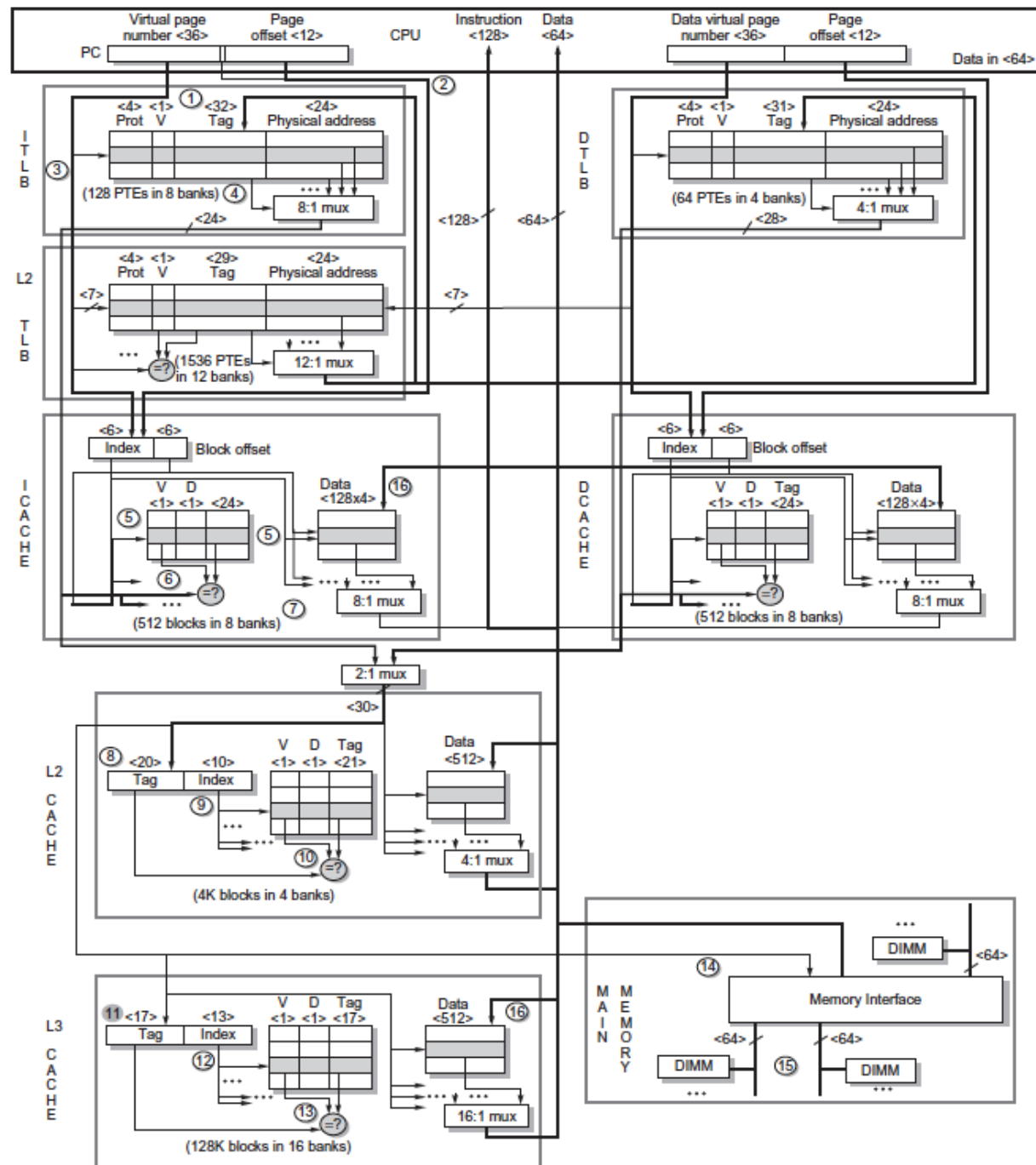
The instruction access path

The data access path

Memory Hierarchies
in the Intel Core i7 6700

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

Virtual Memory & VM

- Virtual Memory

- Virtual Machine
  - VMM / hypervisor

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

Reduce miss rate

Reduce miss penalty

Reduce hit time

- Virtual Memory (the influence of memory organization structure on Cache failure rate)

Homework

1. Consider a two-level memory hierarchy made of L1 and L2 data caches. Assume that both caches use write-back policy on write hit and both have the same block size. List the actions taken in response to the following events:

1） An L1 cache miss when the caches are organized in an inclusive hierarchy.

2） An L1 cache miss when the caches are organized in an exclusive hierarchy.

3） In both parts (1) and (2), consider the possibility that the evicted line might be clean or dirty.

2. Whereas larger caches have lower miss rates, they also tend to have longer hit times. Assume a direct-mapped 8 KB cache has 0.22 ns hit time and miss rate m1; also assume a 4-way associative 64 KB cache has 0.52 ns hit time and a miss rate m2.

1） If the miss penalty is 100 ns, when would it be advantageous to use the smaller cache to reduce the overall memory access time?

2） Repeat part (1) for miss penalties of 10 and 1000 cycles. Conclude when it might be advantageous to use a smaller cache.

knowledge map