Cache Memory

Chapter 17

S. Dandamudi

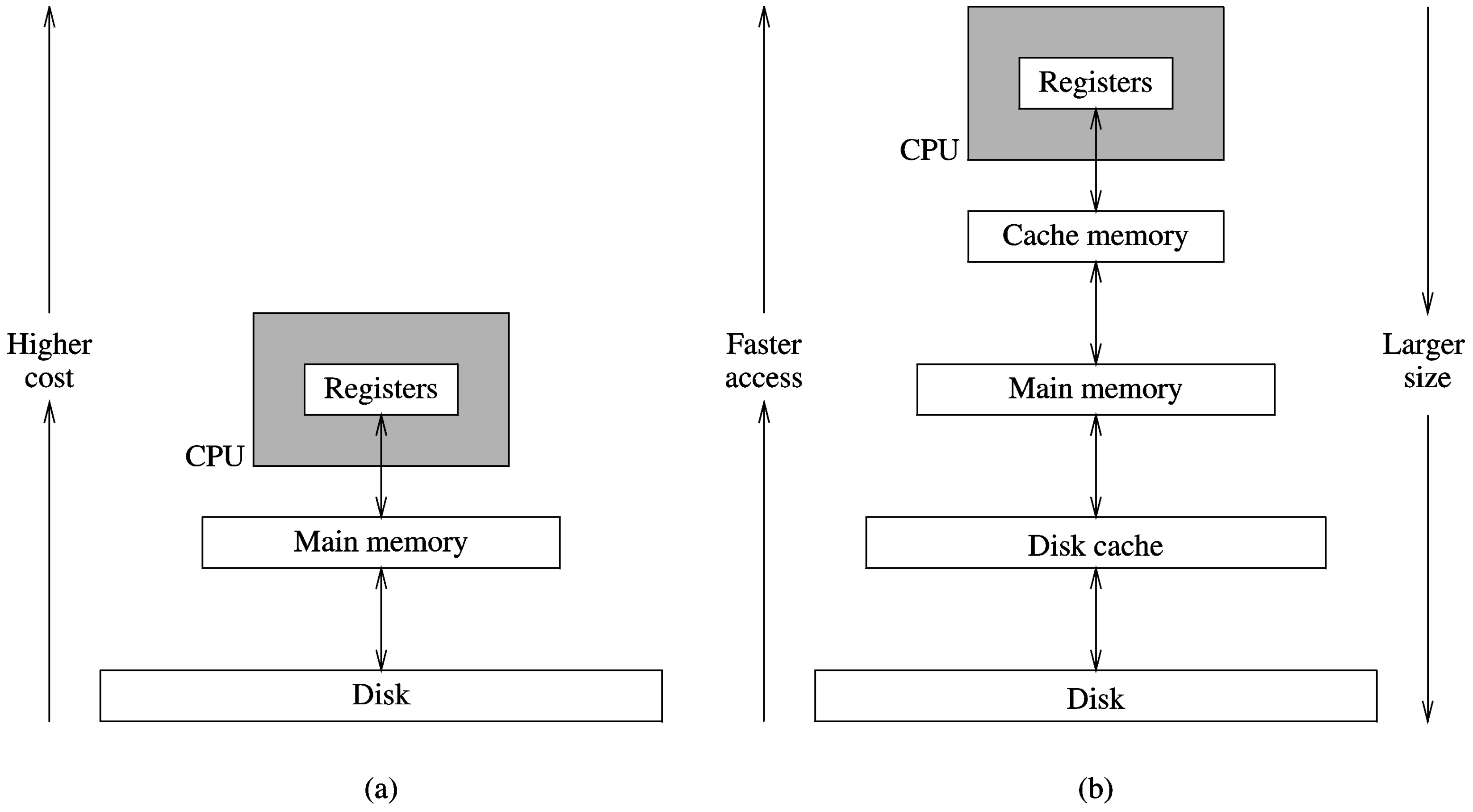
Outline

- Introduction
- How cache memory works
- Why cache memory works
- Cache design basics
- Mapping function
  - Direct mapping
  - Associative mapping
  - Set-associative mapping
- Replacement policies
- Write policies
- Space overhead
- Types of cache misses
- Types of caches
- Example implementations
  - Pentium
  - PowerPC
  - MIPS
- Cache operation summary
- Design issues
  - Cache capacity
  - Cache line size
  - Degree of associatively

Introduction

- Memory hierarchy
  - Registers
  - Memory
  - Disk
  - …
- Cache memory is a small amount of fast memory
  - Placed between two levels of memory hierarchy
    - To bridge the gap in access times
      - Between processor and main memory (our focus)
      - Between main memory and disk (disk cache)
  - Expected to behave like a large amount of fast memory

Introduction (cont'd)

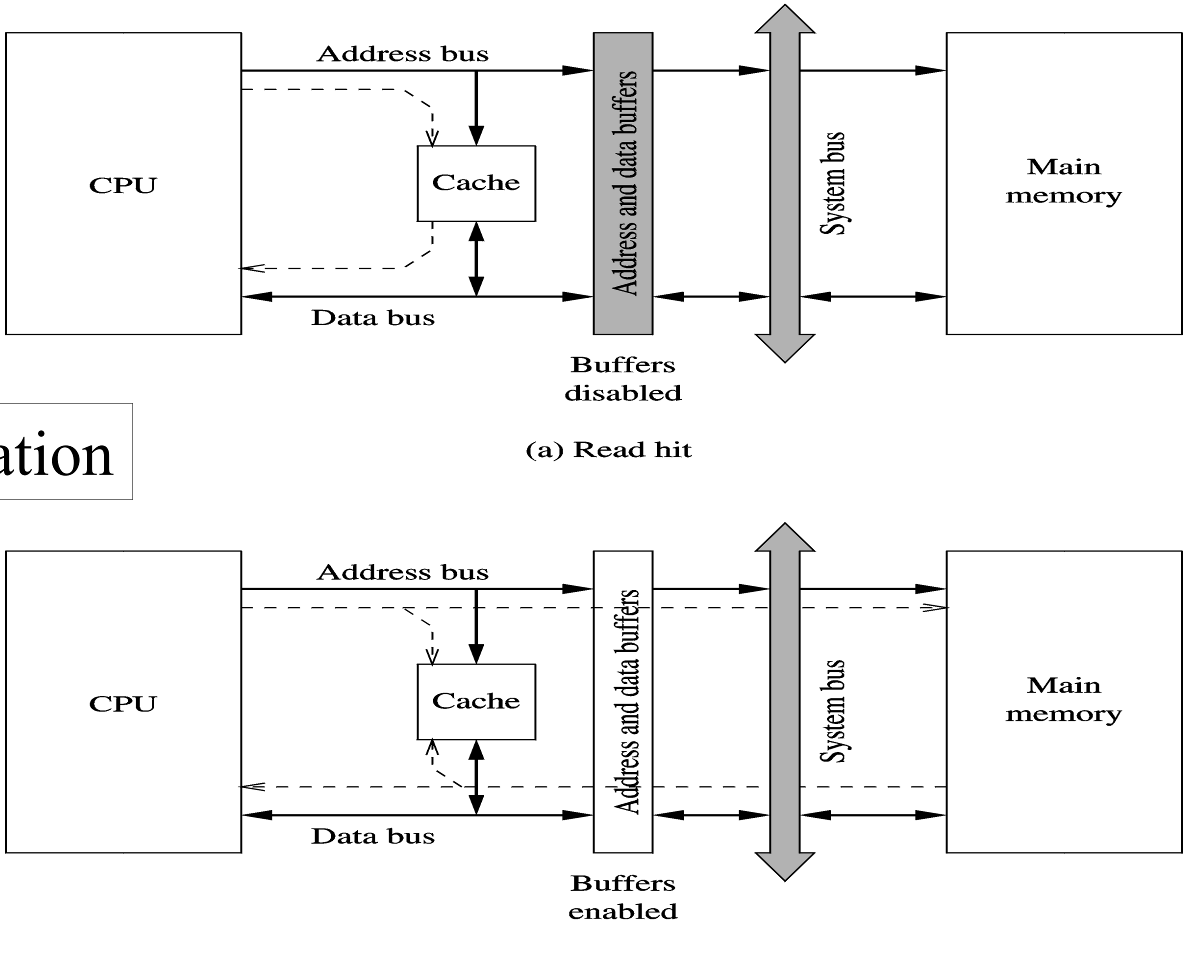


How Cache Memory Works

- Prefetch data into cache before the processor needs it
  - ∗ Need to predict processor future access requirements
    - » Not difficult owing to ***locality of reference***
- Important terms
  - ∗ Miss penalty
  - ∗ Hit ratio
  - ∗ Miss ratio = (1 – hit ratio)
  - ∗ Hit time

How Cache Memory Works (cont'd)

Cache read operation

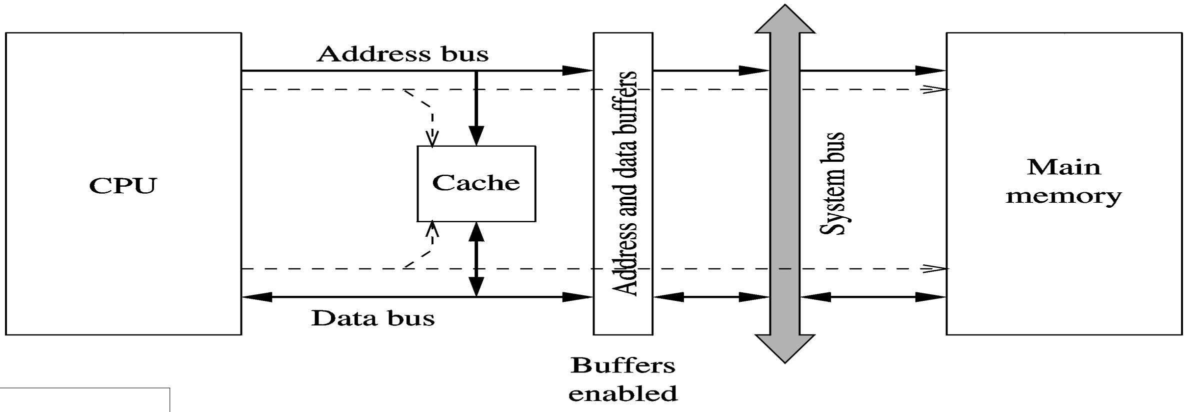


(a) Read hit

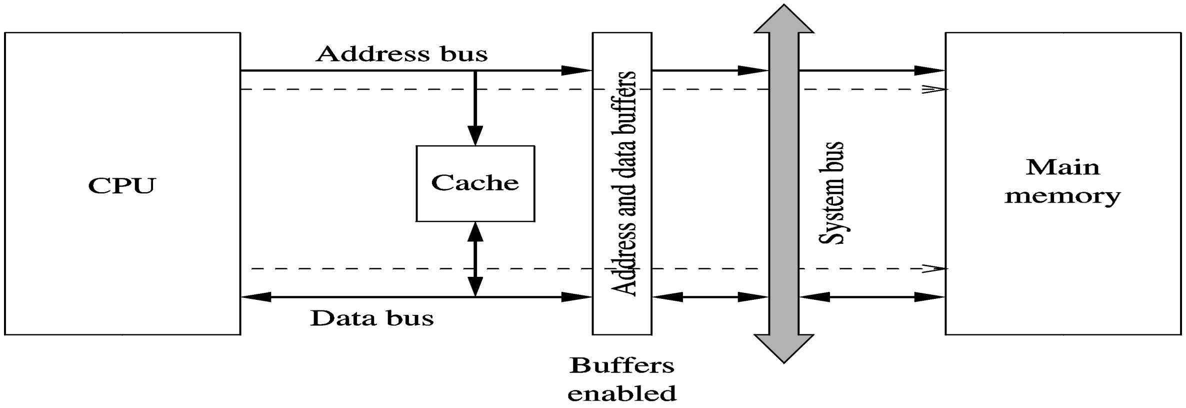
(b) Read miss

How Cache Memory Works (cont'd)

Cache write operation



(a) Write hit



(b) Write miss

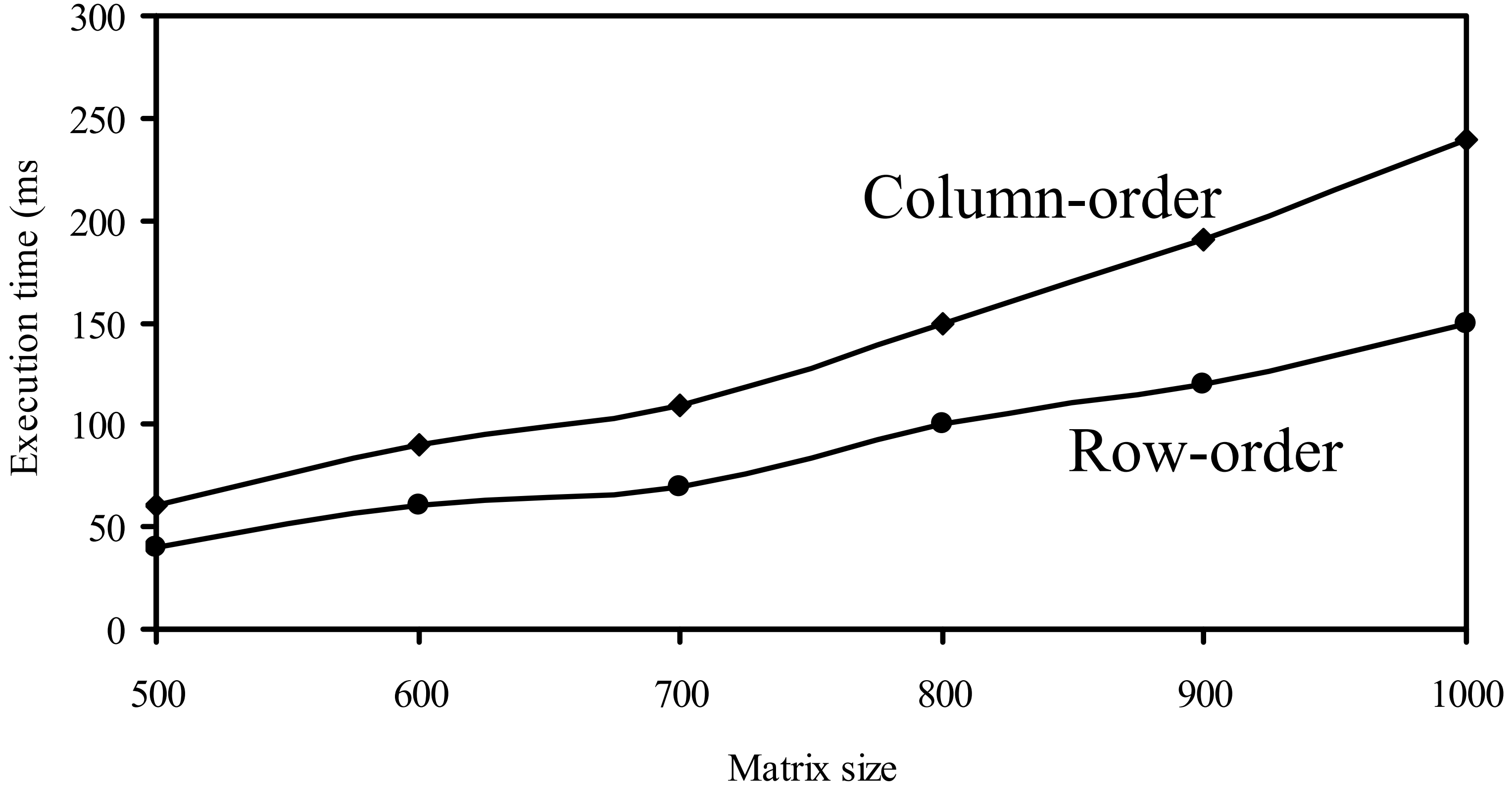
Why Cache Memory Works

- Example

```
for (i=0; i<M; i++)
    for(j=0; j<N; j++)
        X[i][j] = X[i][j] + K;
```

  - ∗ Each element of X is **`double`** (eight bytes)
  - ∗ Loop is executed (M∗N) times
    - » Placing the code in cache avoids access to main memory
      - – Repetitive use (one of the factors)
      - – Temporal locality
    - » Prefetching data
      - – Spatial locality

How Cache Memory Works (cont'd)



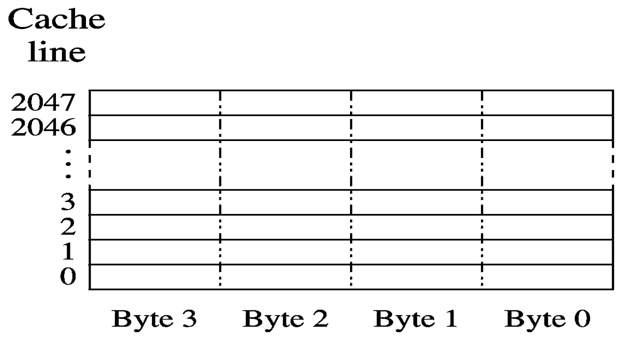
Cache Design Basics

- On every read miss
  - ∗ A fixed number of bytes are transferred
    - » More than what the processor needs
      - – Effective due to spatial locality
- Cache is divided into blocks of ***B*** bytes
    - » ***b***-bits are needed as offset into the block

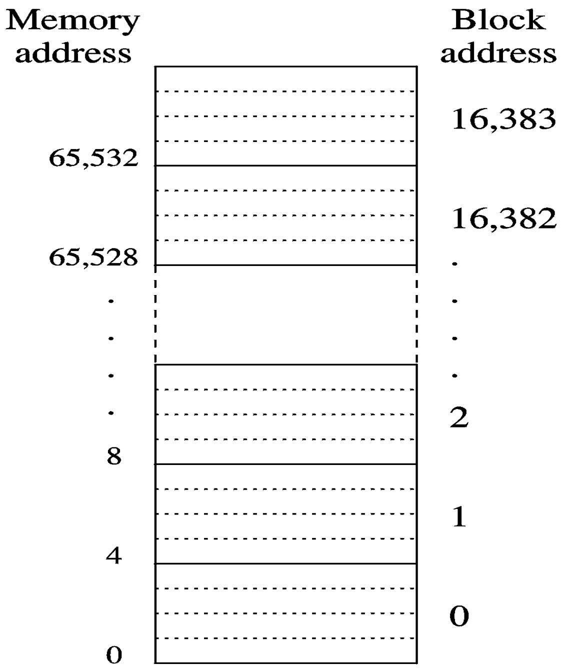
      $\boldsymbol{b} = \log_2 \boldsymbol{B}$
    - » Block are called *cache lines*
- Main memory is also divided into blocks of same size
  - ∗ Address is divided into two parts

Cache Design Basics (cont'd)

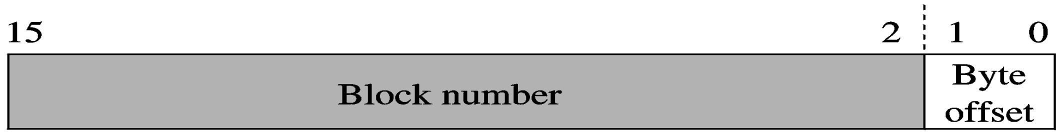
B = 4 bytes
b = 2 bits



Cache memory



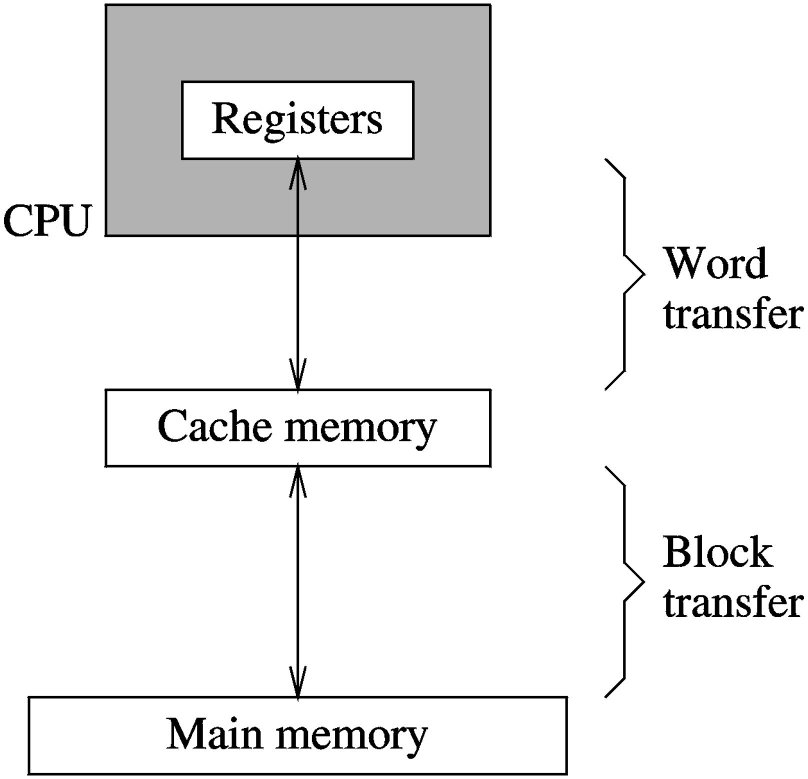
Main memory



Address partition

Cache Design Basics (cont'd)

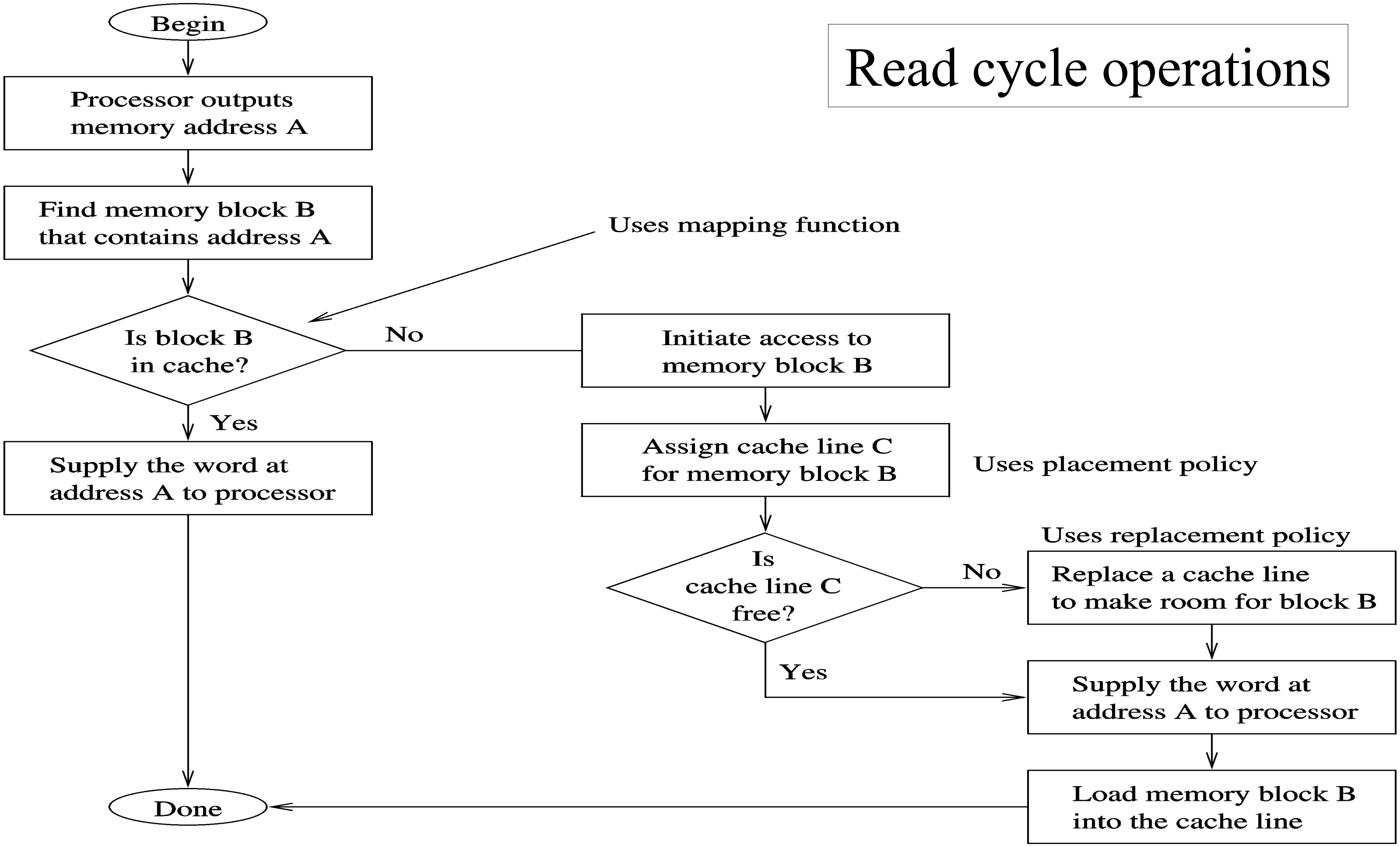
- Transfer between main memory and cache
  - ∗ In units of blocks
  - ∗ Implements spatial locality
- Transfer between main memory and cache
  - ∗ In units of words
- Need policies for
  - ∗ Block placement
  - ∗ Mapping function
  - ∗ Block replacement
  - ∗ Write policies



To be used with S. Dandamudi, "Fundamentals of Computer Organization and Design," Springer, 2003.

Cache Design Basics (cont’d)

Read cycle operations

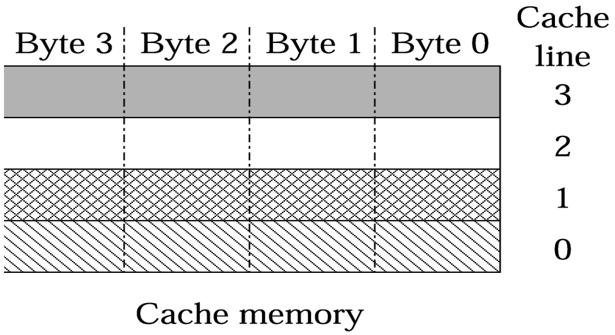

Begin
Processor outputs memory address A
Find memory block B that contains address A
Uses mapping function
Is block B in cache?
No
Yes
Initiate access to memory block B
Supply the word at address A to processor
Assign cache line C for memory block B
Uses placement policy
Uses replacement policy
Is cache line C free?
No
Replace a cache line to make room for block B
Yes
Supply the word at address A to processor
Done
Load memory block B into the cache line

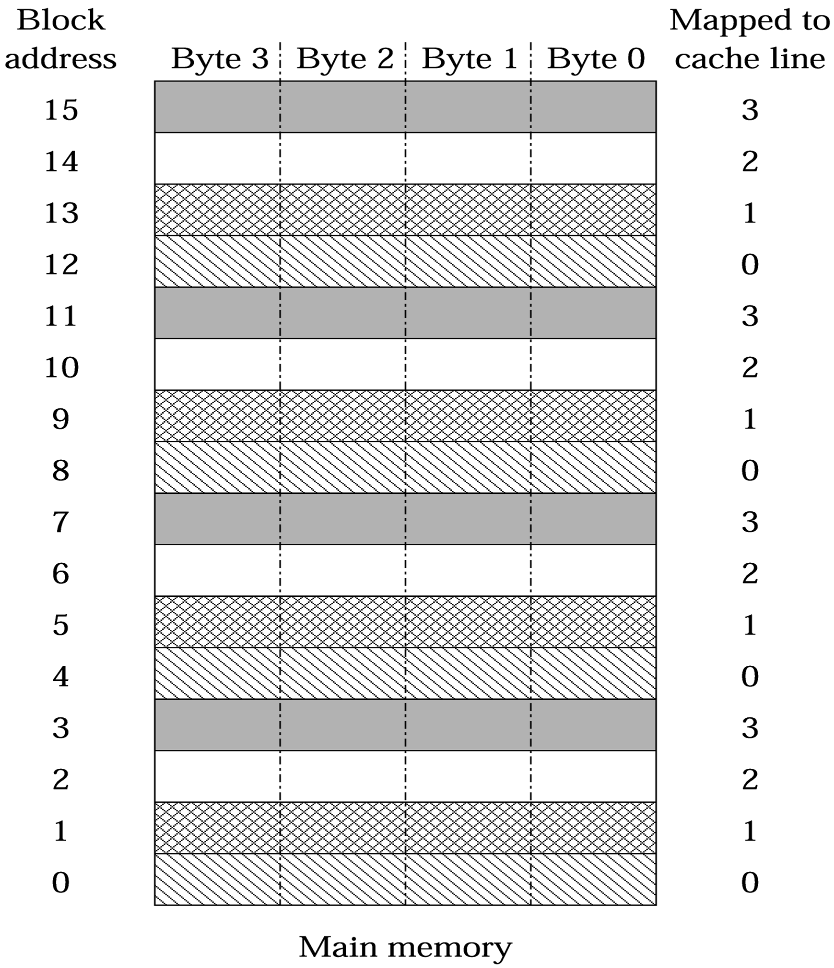
Mapping Function

- Determines how memory blocks are mapped to cache lines
- Three types
  - ∗ Direct mapping
    - » Specifies a single cache line for each memory block
  - ∗ Set-associative mapping
    - » Specifies a set of cache lines for each memory block
  - ∗ Associative mapping
    - » No restrictions
      - – Any cache line can be used for any memory block

Mapping Function (cont'd)

Direct mapping example

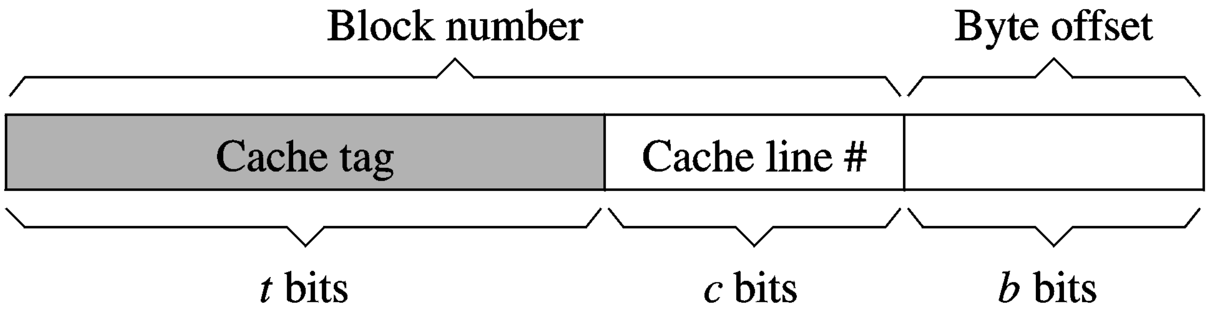




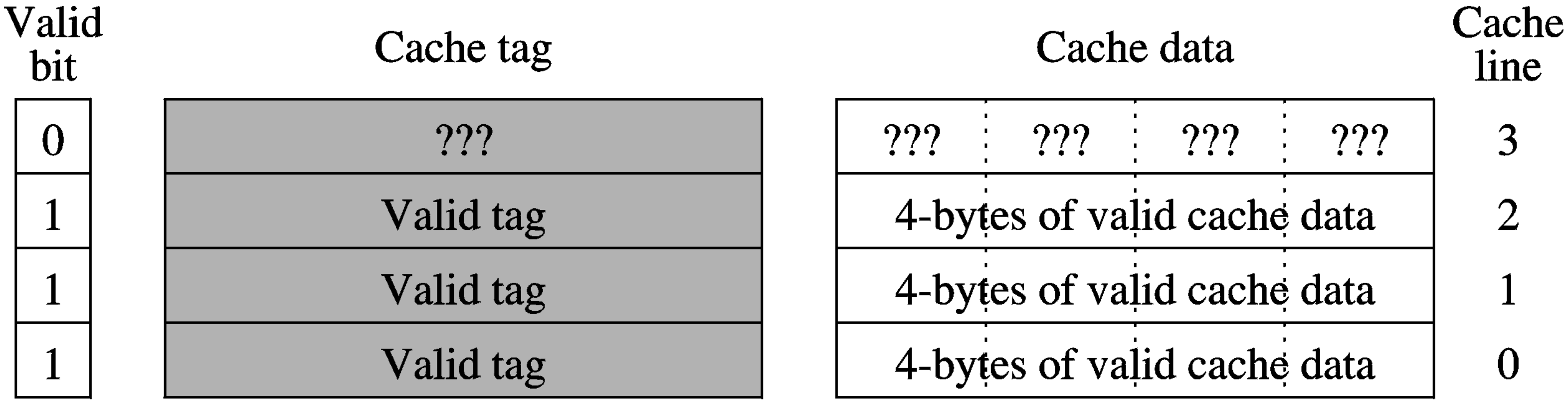
Mapping Function (cont'd)

- Implementing direct mapping
  - ∗ Easier than the other two
  - ∗ Maintains three pieces of information
    - » Cache data
      - – Actual data
    - » Cache tag
      - – Problem: More memory blocks than cache lines
        - ✈Several memory blocks are mapped to a cache line
      - – Tag stores the address of memory block in cache line
    - » Valid bit
      - – Indicates if cache line contains a valid block

Mapping Function (cont'd)



(a) Address partition



Valid bit	Cache tag	Cache data				Cache line
0	???	???	???	???	???	3
1	Valid tag	4-bytes of valid cache data				2
1	Valid tag	4-bytes of valid cache data				1
1	Valid tag	4-bytes of valid cache data				0

(b) Cache memory details

Mapping Function (cont’d)

Direct mapping

Reference pattern:
0, 4, 0, 8, 0, 8,
0, 4, 0, 4, 0, 4

Hit ratio = 0%

Table 17.1 Direct-mapped cache state for Example 17.2

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???

Mapping Function (cont’d)

Direct mapping

Reference pattern:
0, 7, 9, 10, 0, 7,
9, 10, 0, 7, 9, 10

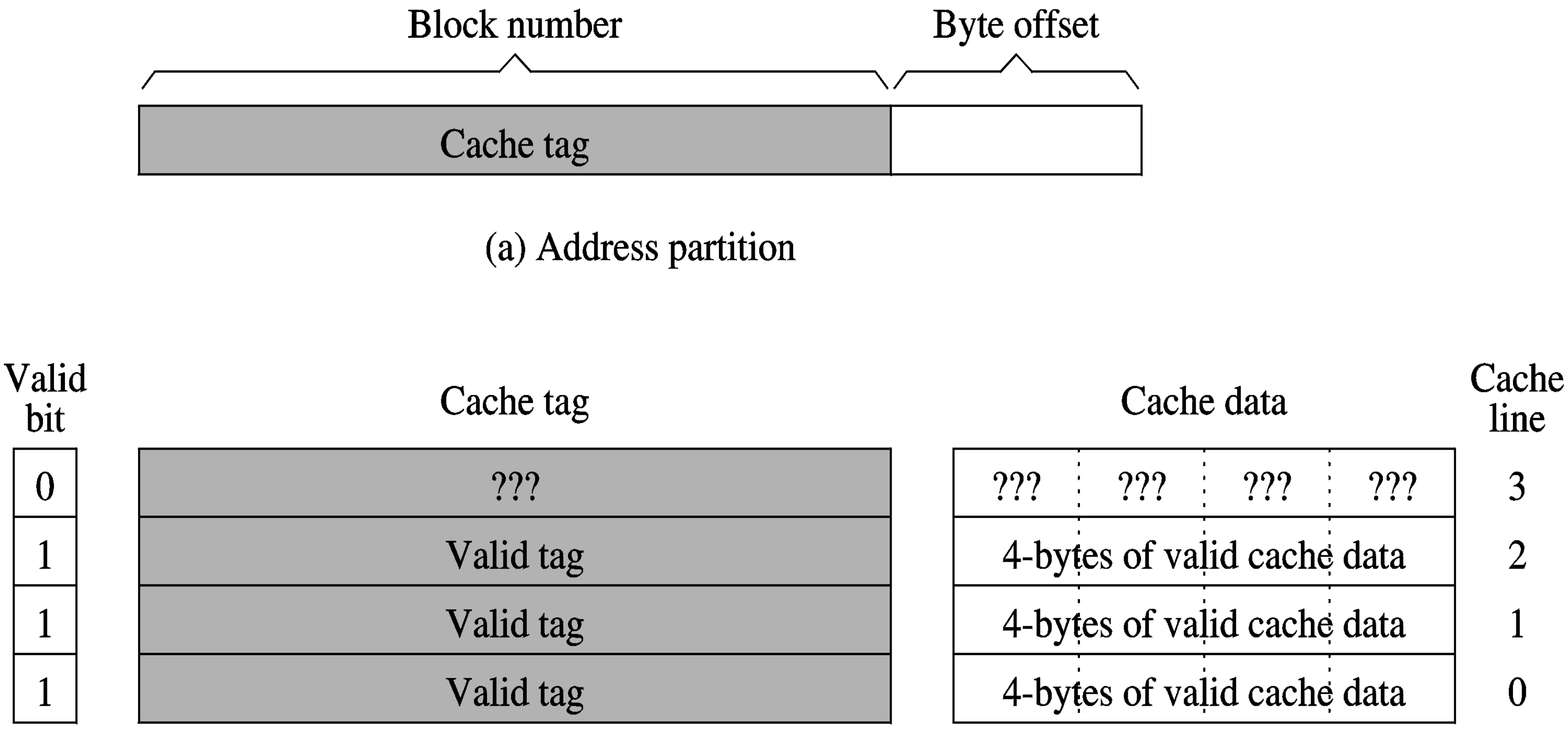
Hit ratio = 67%

Table 17.2 Direct-mapped cache state for Example 17.3

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
7	Miss	Block 0	???	???	Block 7
9	Miss	Block 0	Block 9	???	Block 7
10	Miss	Block 0	Block 9	Block 10	Block 7
0	Hit	Block 0	Block 9	Block 10	Block 7
7	Hit	Block 0	Block 9	Block 10	Block 7
9	Hit	Block 0	Block 9	Block 10	Block 7
10	Hit	Block 0	Block 9	Block 10	Block 7
0	Hit	Block 0	Block 9	Block 10	Block 7
7	Hit	Block 0	Block 9	Block 10	Block 7
9	Hit	Block 0	Block 9	Block 10	Block 7
10	Hit	Block 0	Block 9	Block 10	Block 7

Mapping Function (cont'd)

Associative mapping



(a) Address partition

Valid bit	Cache tag	Cache data				Cache line
0	???	???	???	???	???	3
1	Valid tag	4-bytes of valid cache data				2
1	Valid tag	4-bytes of valid cache data				1
1	Valid tag	4-bytes of valid cache data				0

(b) Cache memory details

Mapping Function (cont'd)

Associative mapping

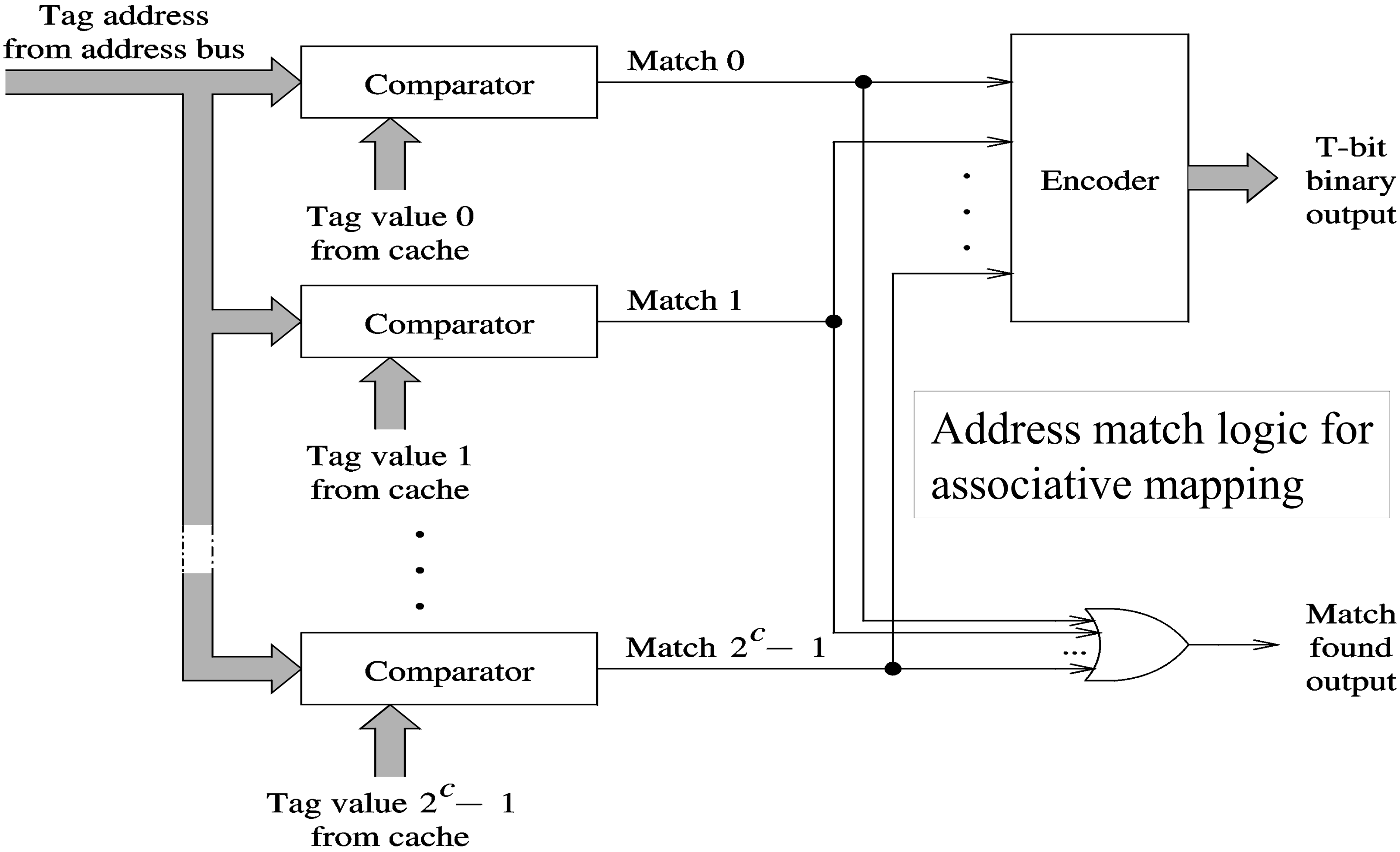
Reference pattern:
0, 4, 0, 8, 0, 8,
0, 4, 0, 4, 0, 4

Hit ratio = 75%

Table 17.3 Fully associative cache state for Example 17.4

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
8	Miss	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
8	Hit	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
4	Hit	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
4	Hit	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
4	Hit	Block 0	Block 4	Block 8	???

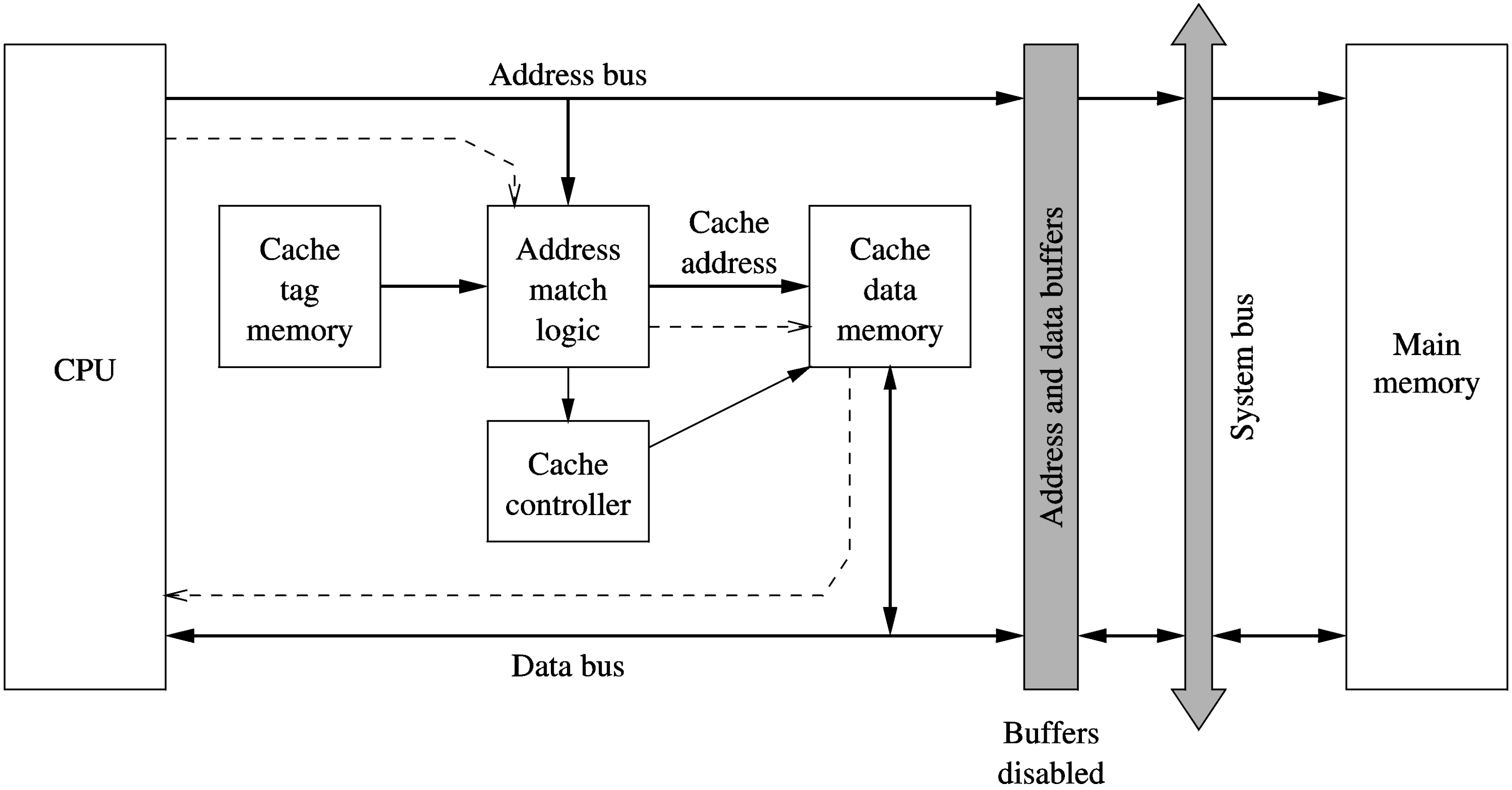
Mapping Function (cont'd)



Address match logic for associative mapping

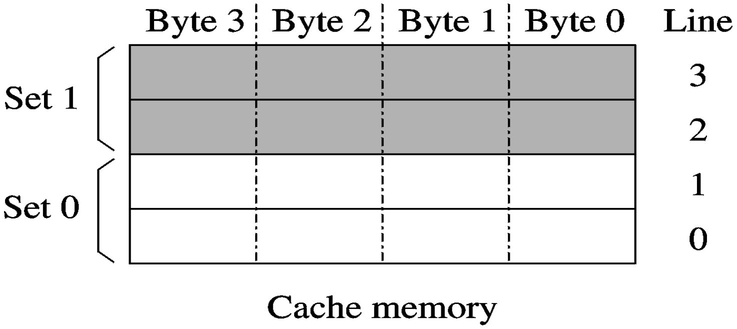
Mapping Function (cont’d)

Associative cache with address match logic



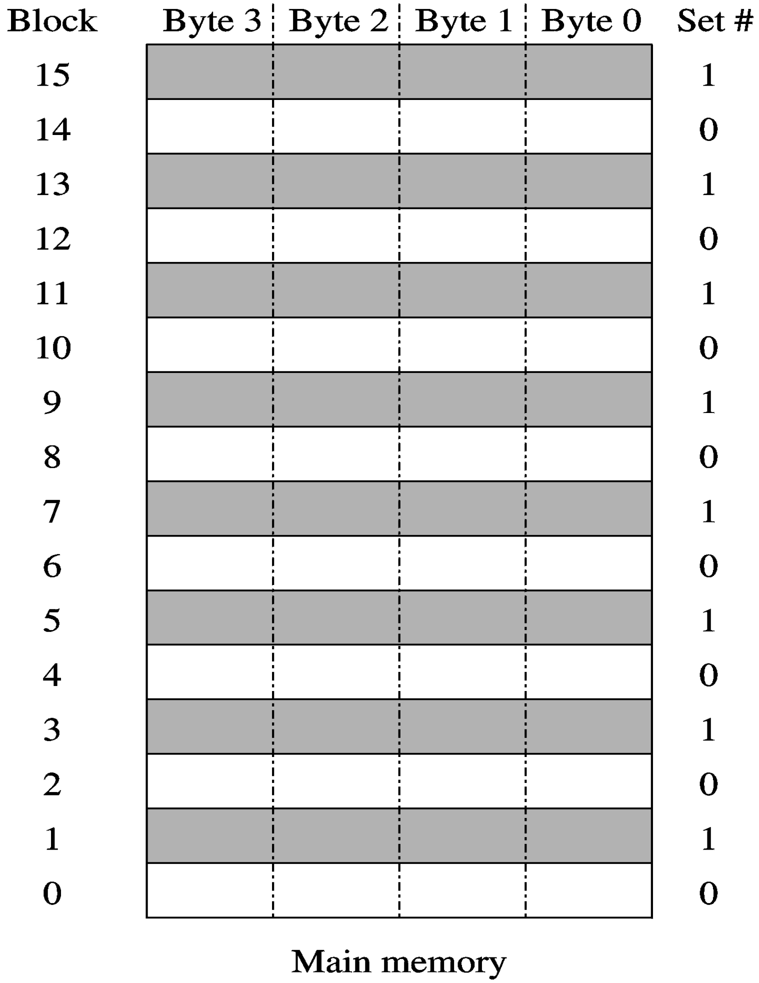
Mapping Function (cont’d)

Set-associative mapping



	Byte 3	Byte 2	Byte 1	Byte 0	Line
Set 1					3
					2
Set 0					1
					0

Cache memory

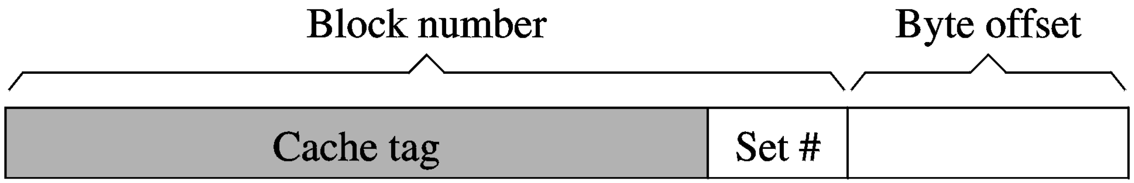


Block	Byte 3	Byte 2	Byte 1	Byte 0	Set #
15					1
14					0
13					1
12					0
11					1
10					0
9					1
8					0
7					1
6					0
5					1
4					0
3					1
2					0
1					1
0					0

Main memory

Mapping Function (cont'd)

Address partition in set-associative mapping

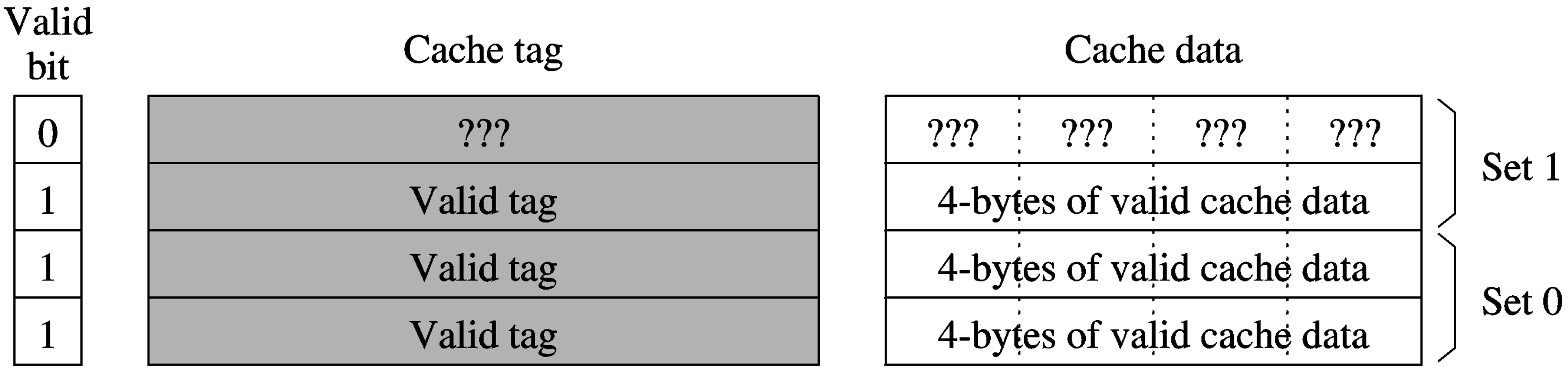


Block number

Byte offset

Cache tag | Set #

(a) Address partition



Valid bit	Cache tag	Cache data	
0	???	??? ??? ??? ???	Set 1
1	Valid tag	4-bytes of valid cache data	
1	Valid tag	4-bytes of valid cache data	Set 0
1	Valid tag	4-bytes of valid cache data	

(b) Cache memory details

Mapping Function (cont’d)

Set-associative mapping

Reference pattern:
0, 4, 0, 8, 0, 8,
0, 4, 0, 4, 0, 4

Hit ratio = 67%

Table 17.4 Set-associative cache state for Example 17.5

Block accessed	Hit or miss	Set 0		Set 1	
		Cache line 0	Cache line 1	Cache line 0	Cache line 1
0	Miss	Block 0	???	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
8	Miss	Block 0	Block 8	???	???
0	Hit	Block 0	Block 8	???	???
8	Hit	Block 0	Block 8	???	???
0	Hit	Block 0	Block 8	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
4	Hit	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
4	Hit	Block 0	Block 4	???	???

Replacement Policies

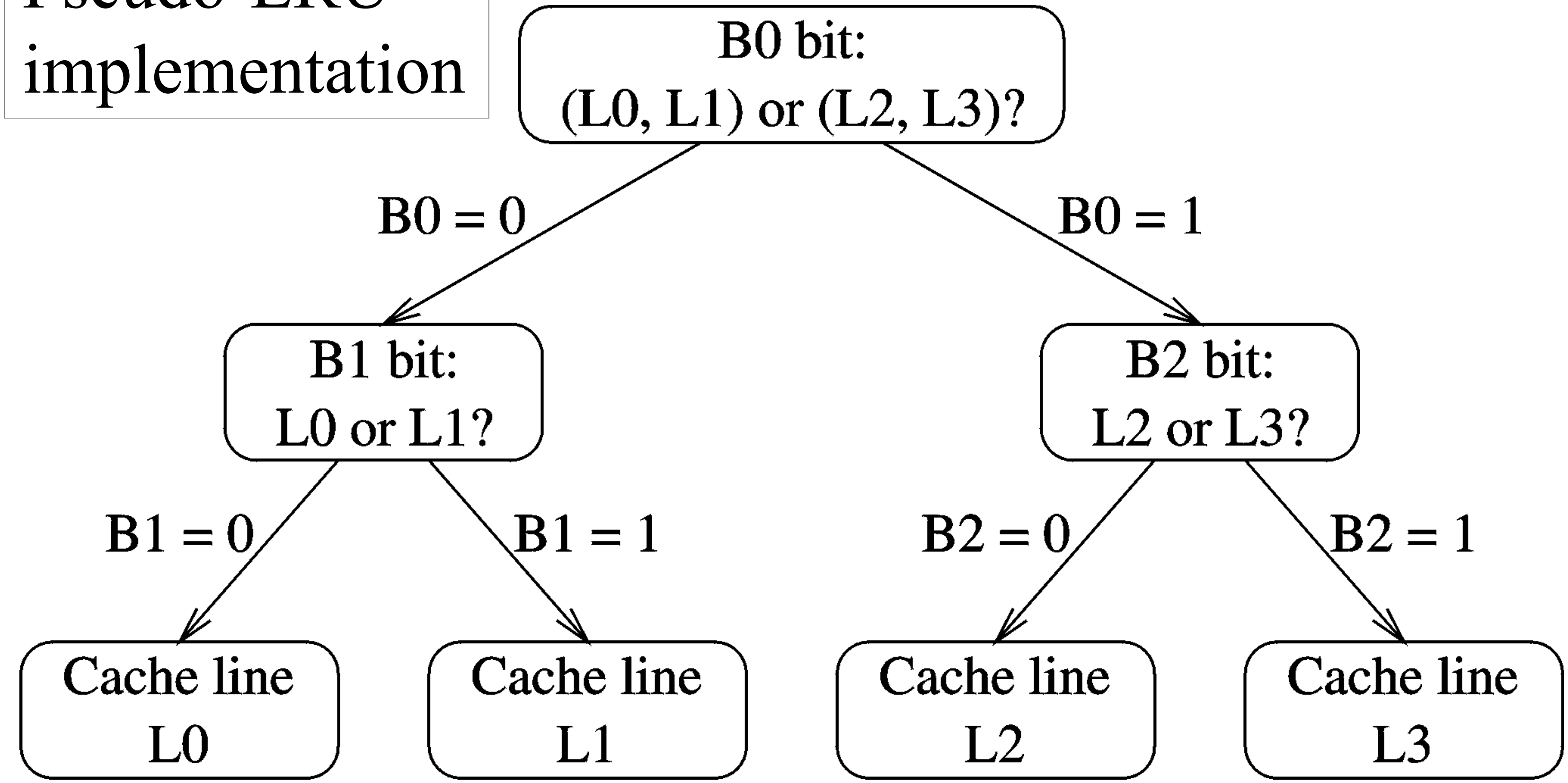
- We invoke the replacement policy
  - ∗ When there is no place in cache to load the memory block
- Depends on the actual placement policy in effect
  - ∗ Direct mapping does not need a special replacement policy
    - » Replace the mapped cache line
  - ∗ Several policies for the other two mapping functions
    - » Popular: LRU (least recently used)
    - » Random replacement
    - » Less interest (FIFO, LFU)

Replacement Policies (cont'd)

- LRU
  - ∗ Expensive to implement
    - » Particularly for set sizes more than four
- Implementations resort to approximation
  - ∗ Pseudo-LRU
    - » Partitions sets into two groups
      - – Maintains the group that has been accessed recently
      - – Requires only one bit
    - » Requires only (***W***-1) bits (***W*** = degree of associativity)
      - – PowerPC is an example
        - ✈Details later

Replacement Policies (cont'd)

Pseudo-LRU implementation



Write Policies

- Memory write requires special attention
  - ∗ We have two copies
    - » A memory copy
    - » A cached copy
  - ∗ Write policy determines how a memory write operation is handled
    - » Two policies
      - – Write-through
        - ✈Update both copies
      - – Write-back
        - ✈Update only the cached copy
        - ✈Needs to be taken care of the memory copy

Write Policies (cont'd)

Cache hit in a write-through cache

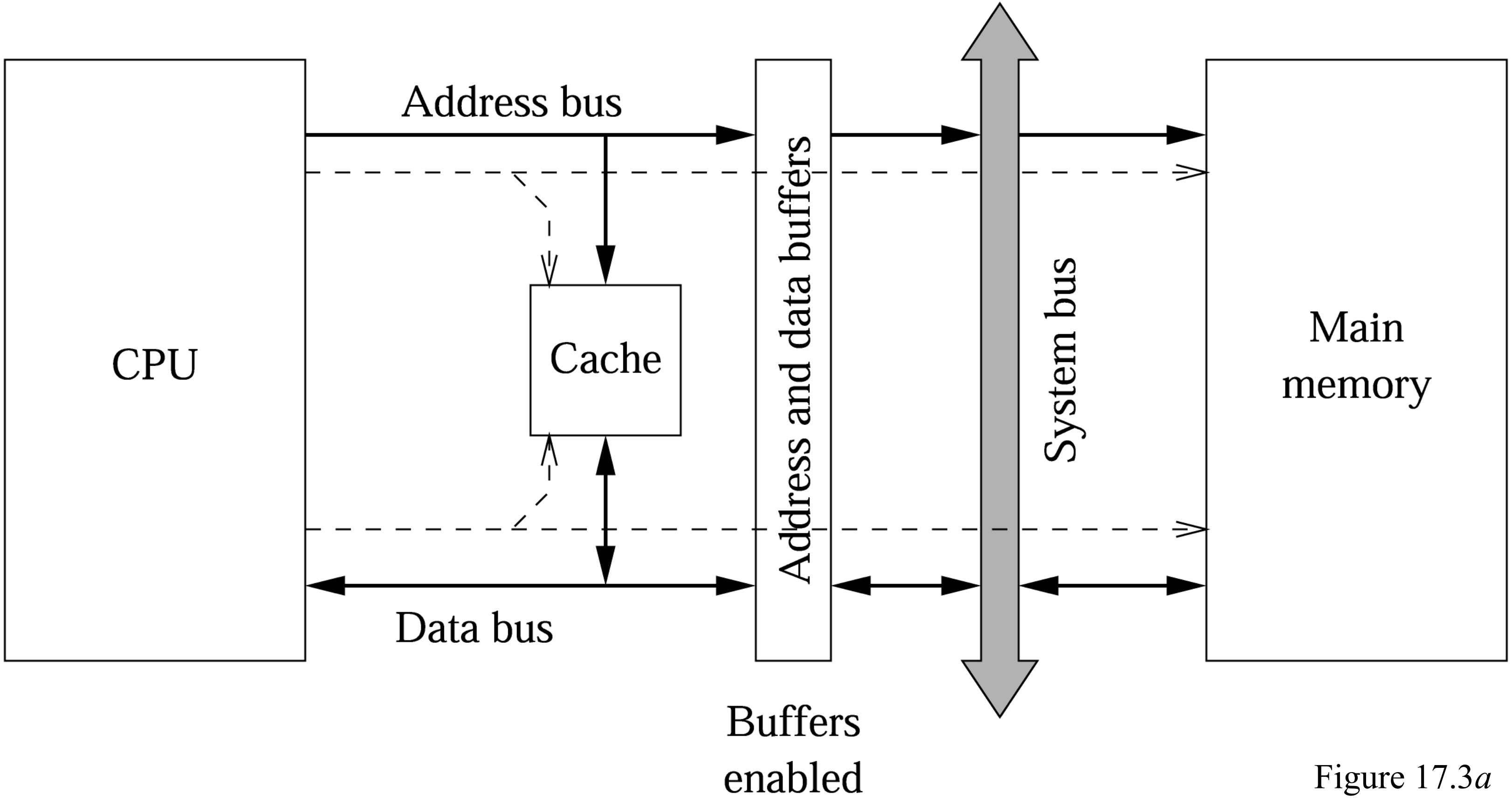
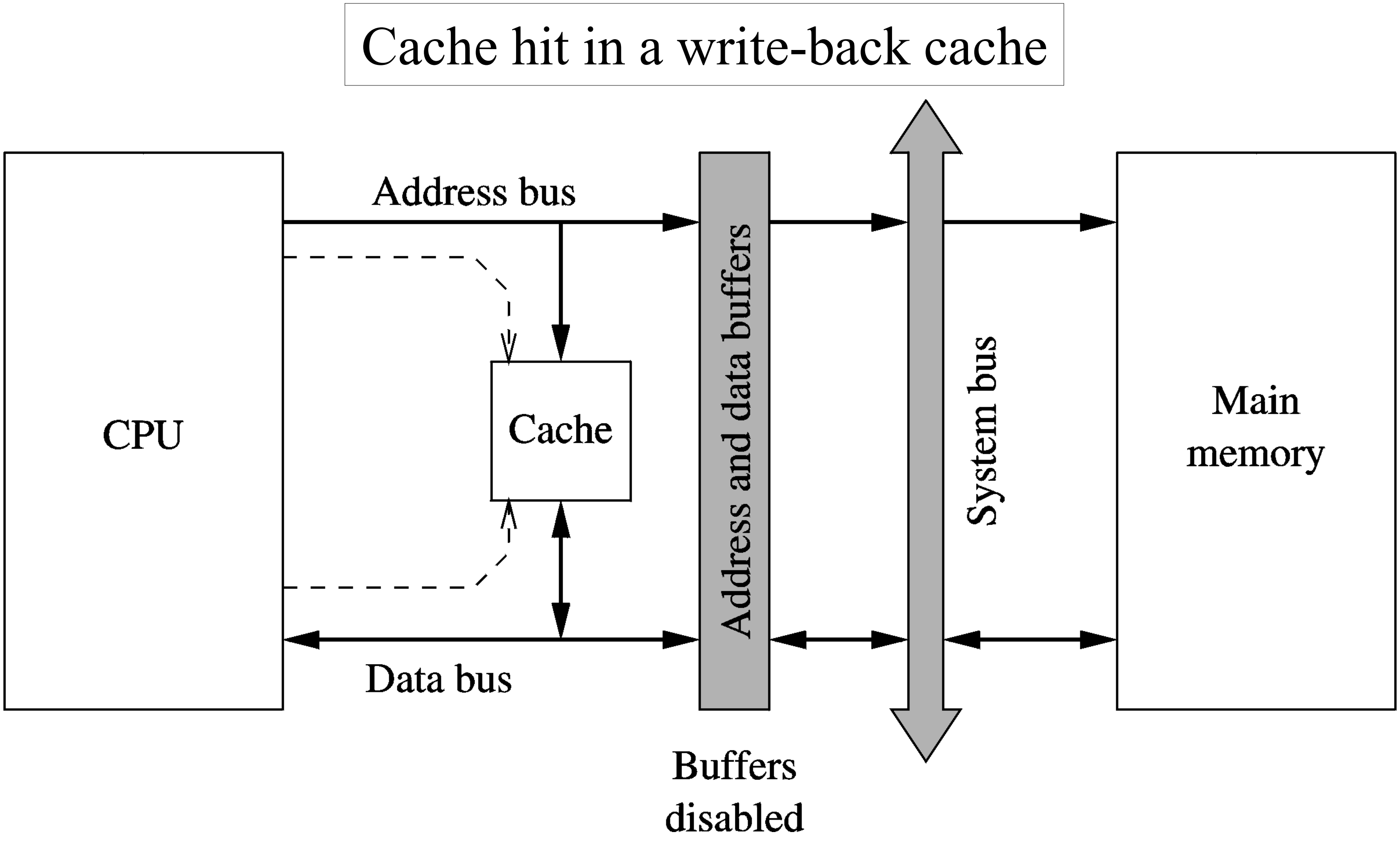


Figure 17.3*a*

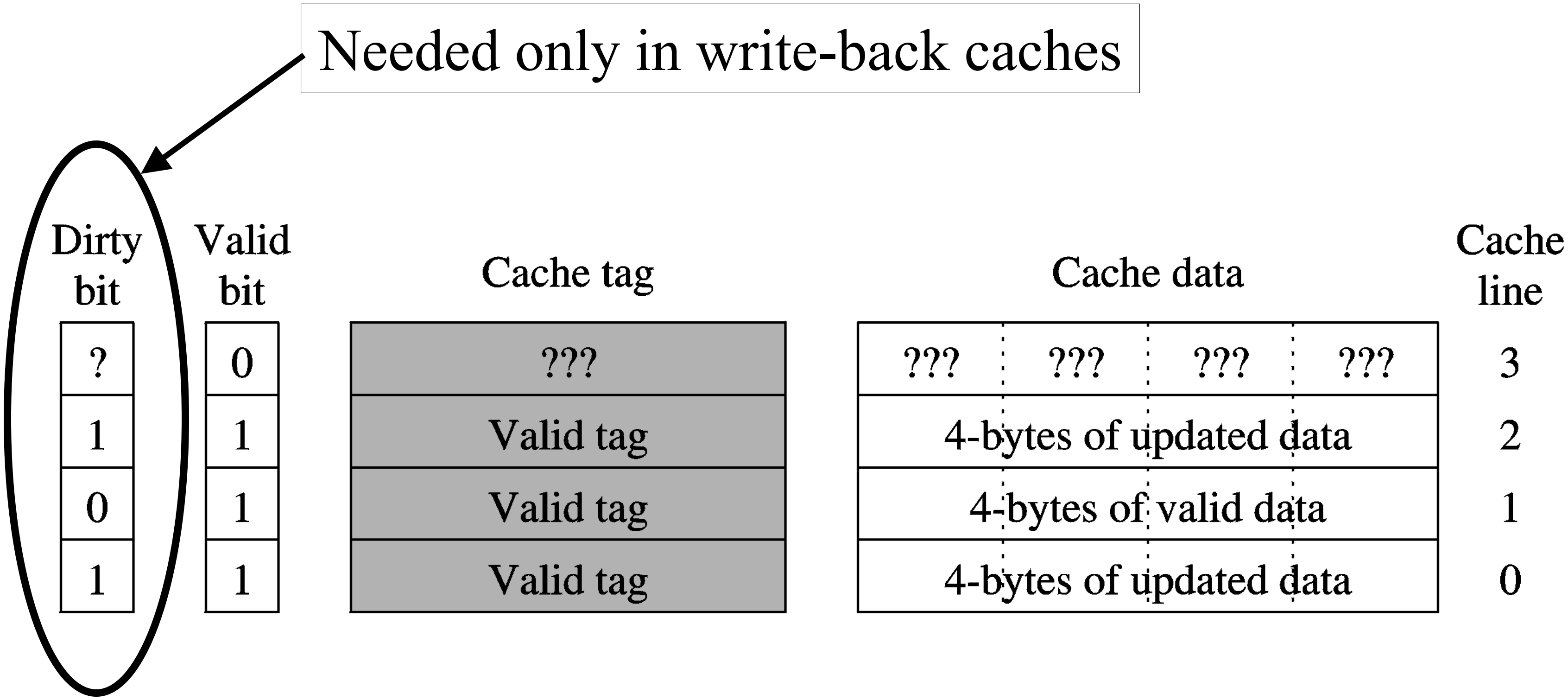
Write Policies (cont'd)



Write Policies (cont'd)

- Write-back policy
  - ∗ Updates the memory copy when the cache copy is being replaced
    - » We first write the cache copy to update the memory copy
  - ∗ Number of write-backs can be reduced if we write only when the cache copy is different from memory copy
    - » Done by associating a ***dirty bit*** or ***update bit***
      - – Write back only when the dirty bit is 1
    - » Write-back caches thus require two bits
      - – A valid bit
      - – A dirty or update bit

Write Policies (cont'd)



Needed only in write-back caches

Dirty bit	Valid bit	Cache tag	Cache data	Cache line
?	0	???	??? ??? ??? ???	3
1	1	Valid tag	4-bytes of updated data	2
0	1	Valid tag	4-bytes of valid data	1
1	1	Valid tag	4-bytes of updated data	0

Write Policies (cont'd)

- Other ways to reduce write traffic
  - ∗ Buffered writes
    - » Especially useful for write-through policies
    - » Writes to memory are buffered and written at a later time
      - – Allows write combining
        - ✈Catches multiple writes in the buffer itself
  - ∗ Example: Pentium
    - » Uses a 32-byte write buffer
    - » Buffer is written at several trigger points
      - – An example trigger point
        - ✈Buffer full condition

Write Policies (cont'd)

- Write-through versus write-back
  - ∗ Write-through
    - » ***Advantage***
      - – Both cache and memory copies are consistent
        - ✈Important in multiprocessor systems
    - » ***Disadvantage***
      - – Tends to waste bus and memory bandwidth
  - ∗ Write-back
    - » ***Advantage***
      - – Reduces write traffic to memory
    - » ***Disadvantages***
      - – Takes longer to load new cache lines
      - – Requires additional dirty bit

Space Overhead

- The three mapping functions introduce different space overheads
  - ∗ Overhead decreases with increasing degree of associativity
    - » Several examples in the text

4 GB address space
32 KB cache

Table 17.5 Cache space overhead for the three organizations

Block size	Direct mapping (%)	4-way set-associative (%)	Fully associative (%)
32 bytes	7	7.8	11
4 bytes	56	62.5	97

Types of Cache Misses

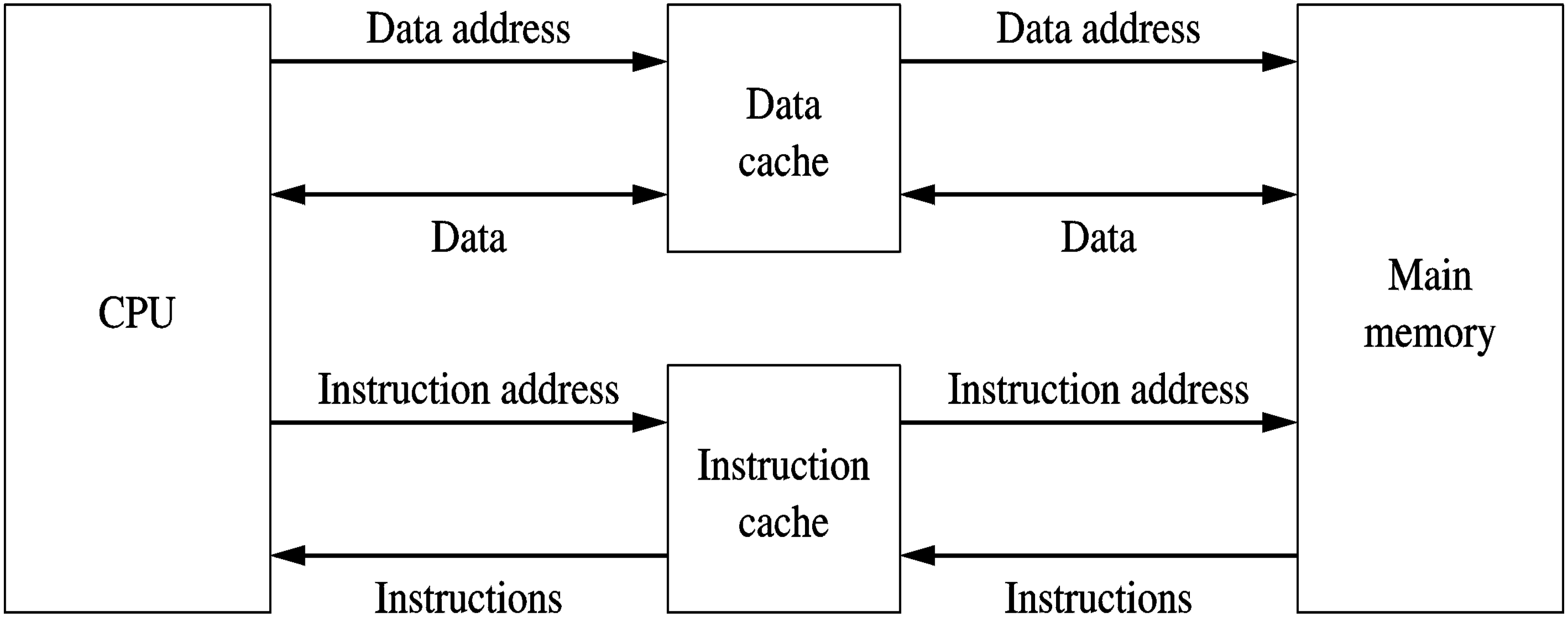
- Three types
  - ∗ Compulsory misses
    - » Due to first-time access to a block
      - – Also called ***cold-start misses*** or ***compulsory line fills***
  - ∗ Capacity misses
    - » Induced due to cache capacity limitation
    - » Can be avoided by increasing cache size
  - ∗ Conflict misses
    - » Due to conflicts caused by direct and set-associative mappings
      - – Can be completely eliminated by fully associative mapping
      - – Also called ***collision misses***

Types of Cache Misses (cont'd)

- Compulsory misses
  - ∗ Reduced by increasing block size
    - » We prefetch more
    - » Cannot increase beyond a limit
      - – Cache misses increase
- Capacity misses
  - ∗ Reduced by increasing cache size
    - » Law of diminishing returns
- Conflict misses
  - ∗ Reduced by increasing degree of associativity
    - » Fully associative mapping: no conflict misses

Types of Caches

- Separate instruction and data caches
  - » Initial cache designs used unified caches
  - » Current trend is to use separate caches (for level 1)

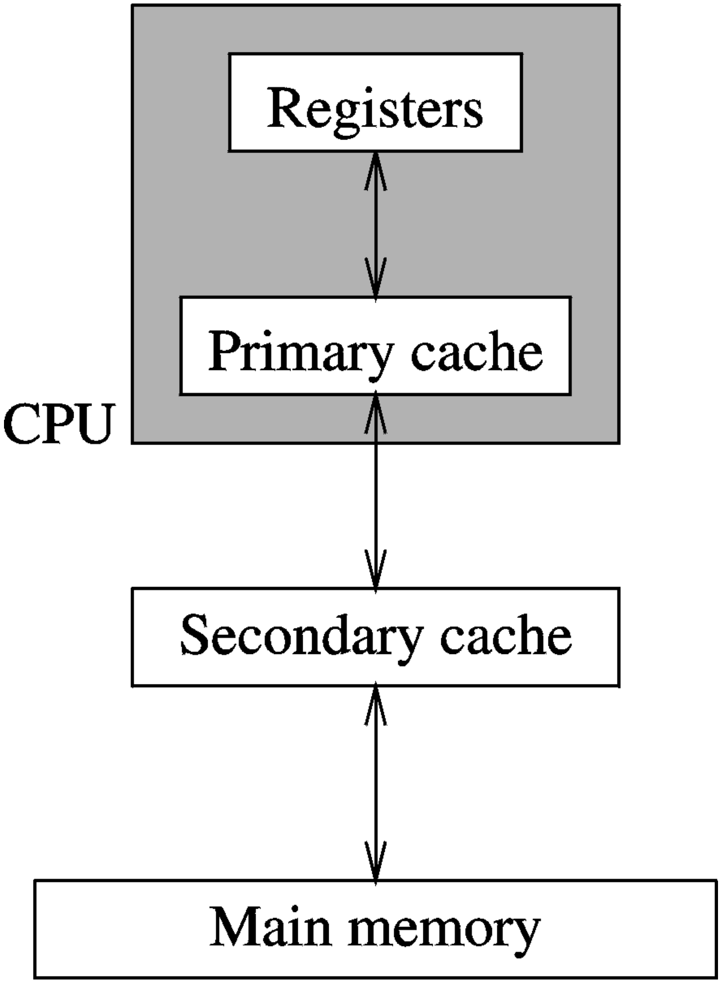


Types of Caches (cont'd)

- Several reasons for preferring separate caches
  - ∗ Locality tends to be stronger
  - ∗ Can use different designs for data and instruction caches
    - » Instruction caches
      - – Read only, dominant sequential access
      - – No need for write policies
      - – Can use a simple direct mapped cache implementation
    - » Data caches
      - – Can use a set-associative cache
      - – Appropriate write policy can be implemented
  - ∗ Disadvantage
    - » Rigid boundaries between data and instruction caches

Types of Caches (cont'd)

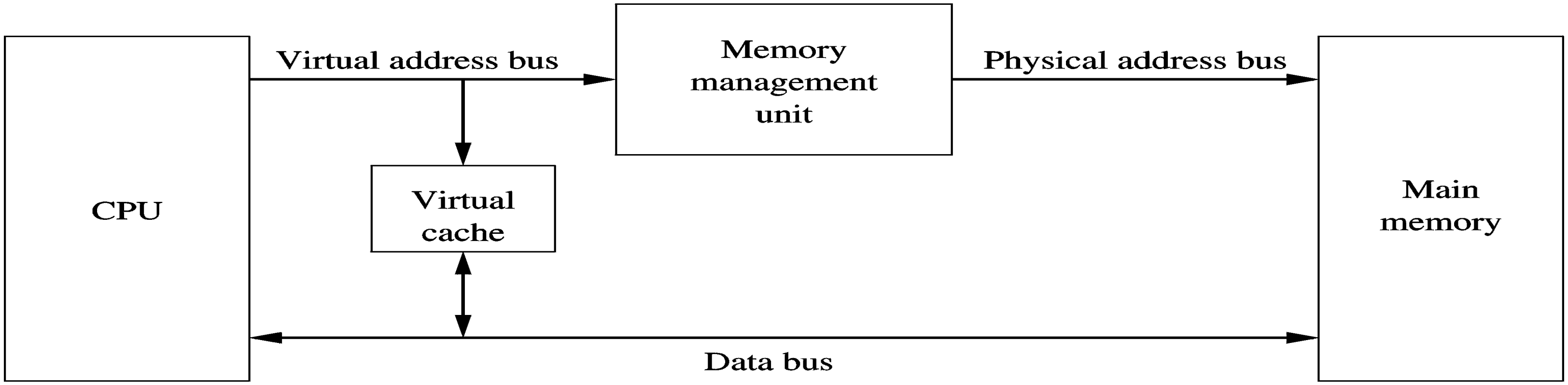
- Number of cache levels
  - ∗ Most use two levels
    - » Primary (level 1 or L1)
      - – On-chip
    - » Secondary (level 2 or L2)
      - – Off-chip
  - ∗ Examples
    - » Pentium
      - – L1: 32 KB
      - – L2: up to 2 MB
    - » PowerPC
      - – L1: 64 KB
      - – L2: up to 1 MB



Types of Caches (cont'd)

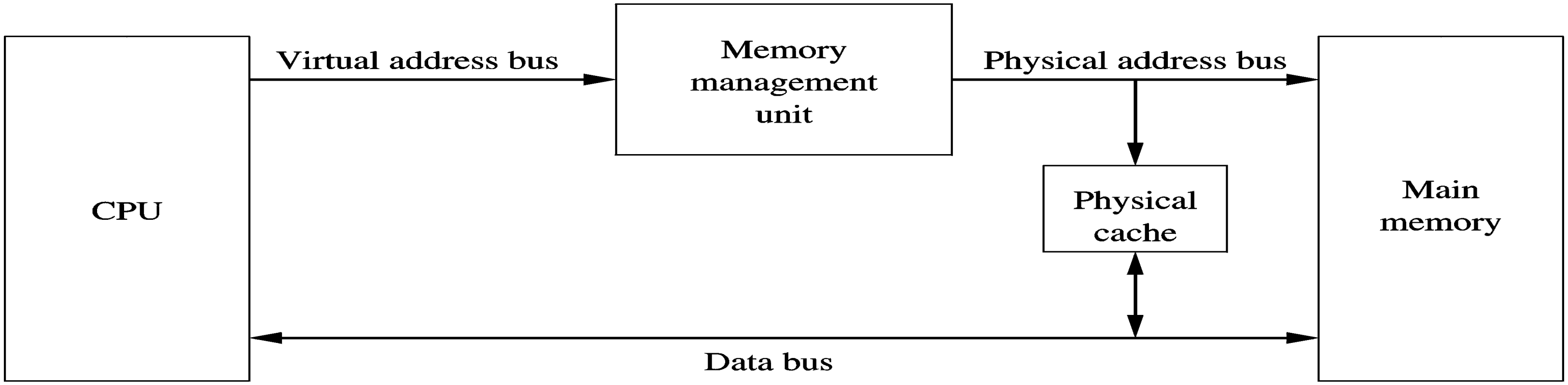
- Two-level caches work as follows:
  - ∗ First attempts to get data from L1 cache
    - » If present in L1, gets data from L1 cache ("L1 cache hit")
    - » If not, data must come form L2 cache or main memory ("L1 cache miss")
  - ∗ In case of L1 cache miss, tries to get from L2 cache
    - » If data are in L2, gets data from L2 cache ("L2 cache hit")
      - Data block is written to L1 cache
    - » If not, data comes from main memory ("L2 cache miss")
      - Main memory block is written into L1 and L2 caches
- Variations on this basic scheme are possible

Types of Caches (cont'd)



(a) Virtual cache

Virtual and physical caches



(b) Physical cache

Example Implementations

- We look at three processors
  - ∗ Pentium
  - ∗ PowerPC
  - ∗ MIPS
- Pentium implementation
  - ∗ Two levels
    - » L1 cache
      - – Split cache design
        - ✈Separate data and instruction caches
    - » L2 cache
      - – Unified cache design

Example Implementations (cont'd)

- Pentium allows each page/memory region to have its own caching attributes
  - Uncacheable
    - All reads and writes go directly to the main memory
      - Useful for
        - Memory-mapped I/O devices
        - Large data structures that are read once
        - Write-only data structures
  - Write combining
    - Not cached
    - Writes are buffered to reduce access to main memory
      - Useful for video buffer frames

Example Implementations (cont'd)

- Write-through
  - Uses write-through policy
  - Writes are delayed as they go though a write buffer as in write combining mode
- Write back
  - Uses write-back policy
  - Writes are delayed as in the write-through mode
- Write protected
  - Inhibits cache writes
  - Write are done directly on the memory

Example Implementations (cont'd)

- Two bits in control register CR0 determine the mode
  - Cache disable (CD) bit
  - w Not write-through (NW) bit

Table 17.6 Pentium family cache operating modes

CD	NW	Write policy	Read miss	Write miss
0	0	Write-through	Cache line filled	Cache line filled
0	1	Invalid combination—causes exception		
1	0	Write-through	No cache line fills	No cache line fills
1	1	Write-back	No cache line fills	No cache line fills

Example Implementations (cont'd)

PowerPC cache implementation

- Two levels
  - » L1 cache
    - – Split cache
      - ✈Each: 32 KB eight-way associative
    - – Uses pseudo-LRU replacement
    - – Instruction cache: read-only
    - – Data cache: read/write
      - ✈Choice of write-through or write-back
  - » L2 cache
    - – Unified cache as in Pentium
    - – Two-way set associative

Example Implementations (cont'd)

- Write policy type and caching attributes can be set by OS at the block or page level
- L2 cache requires only a single bit to implement LRU
  - » Because it is 2-way associative
- L1 cache implements a pseudo-LRU
  - » Each set maintains seven PLRU bits (B0–B6)

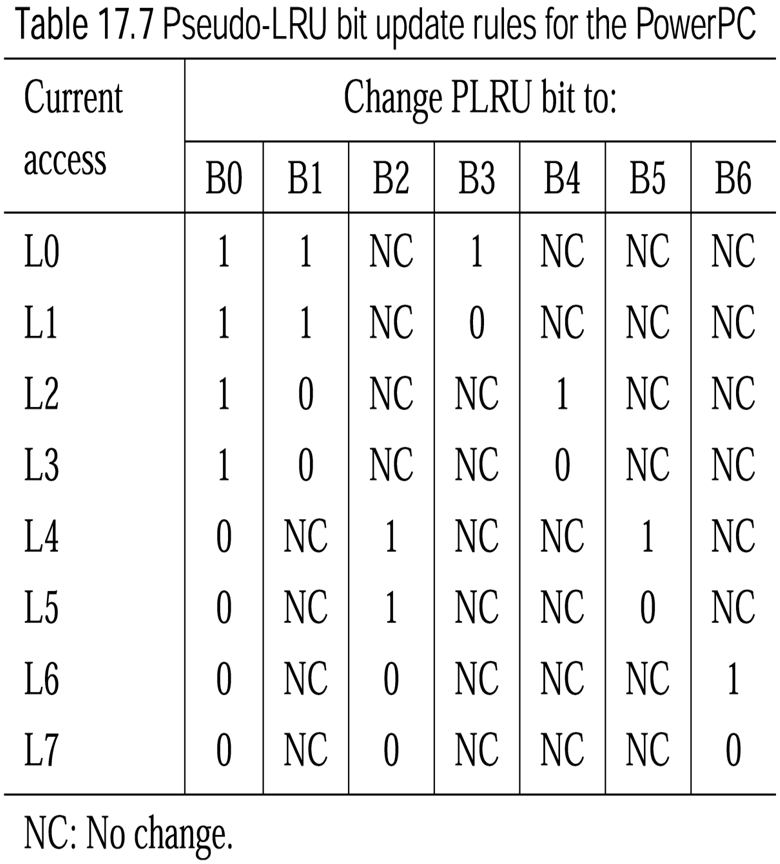


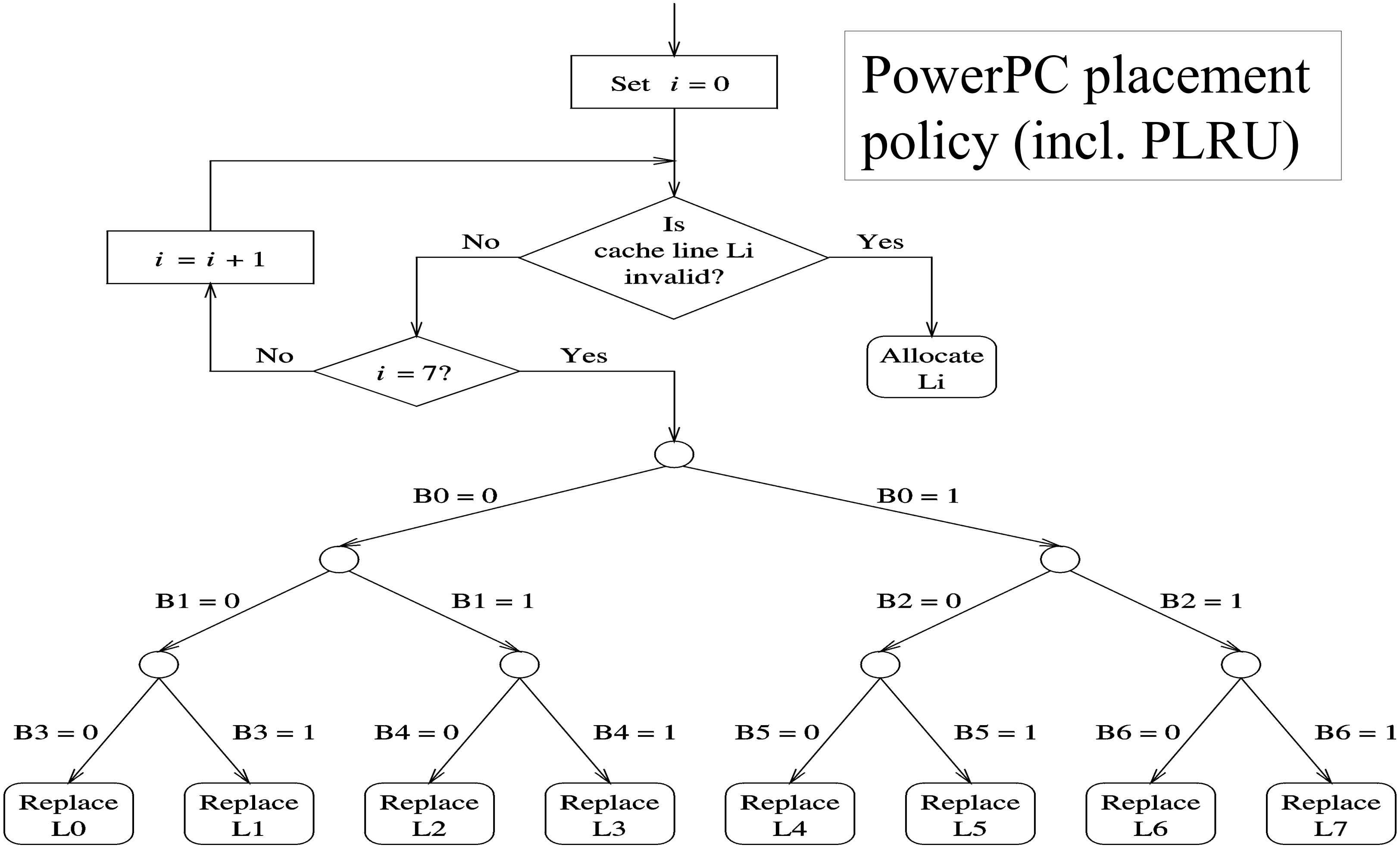
Table 17.7 Pseudo-LRU bit update rules for the PowerPC

Current access	Change PLRU bit to:						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	NC	1	NC	NC	NC
L1	1	1	NC	0	NC	NC	NC
L2	1	0	NC	NC	1	NC	NC
L3	1	0	NC	NC	0	NC	NC
L4	0	NC	1	NC	NC	1	NC
L5	0	NC	1	NC	NC	0	NC
L6	0	NC	0	NC	NC	NC	1
L7	0	NC	0	NC	NC	NC	0

NC: No change.

Example Implementations (cont’d)

PowerPC placement policy (incl. PLRU)



Example Implementations (cont'd)

MIPS implementation

- Two-level cache
  - L1 cache
    - Split organization
    - Instruction cache
      - Virtual cache
      - Direct mapped
      - Read-only
    - Data cache
      - Virtual cache
      - Direct mapped
      - Uses write-back policy

L1 line size: 16 or 32 bytes

Example Implementations (cont'd)

- » L2 cache
  - – Physical cache
  - – Either unified or split
    - ✈Configured at boot time
  - – Direct mapped
  - – Uses write-back policy
  - – Cache block size
    - ✈16, 32, 64, or 128 bytes
    - ✈Set at boot time
  - – L1 cache line size ≤ L2 cache size

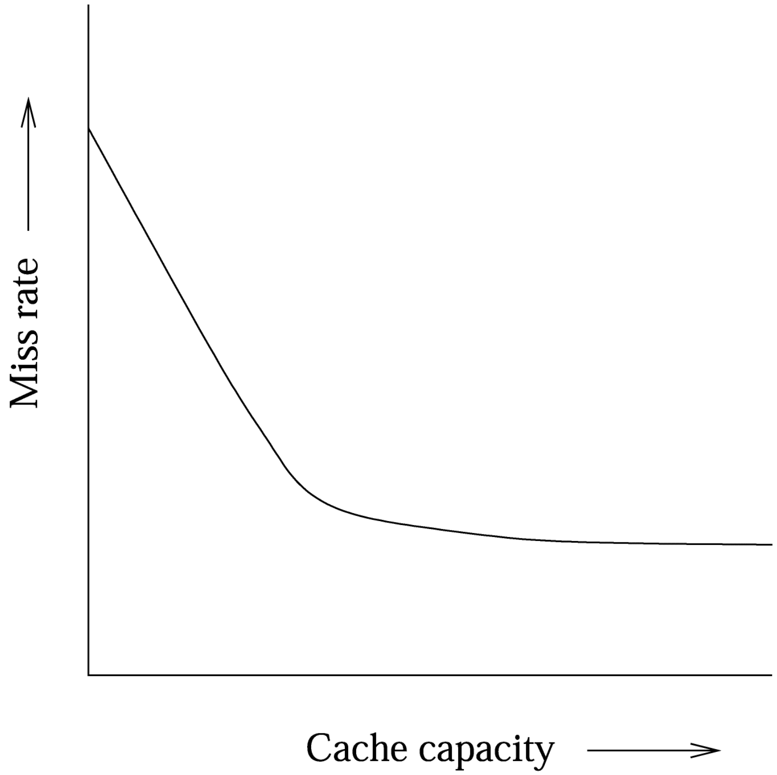
- * Direct mapping simplifies replacement
  - » No need for LRU type complex implementation

Cache Operation Summary

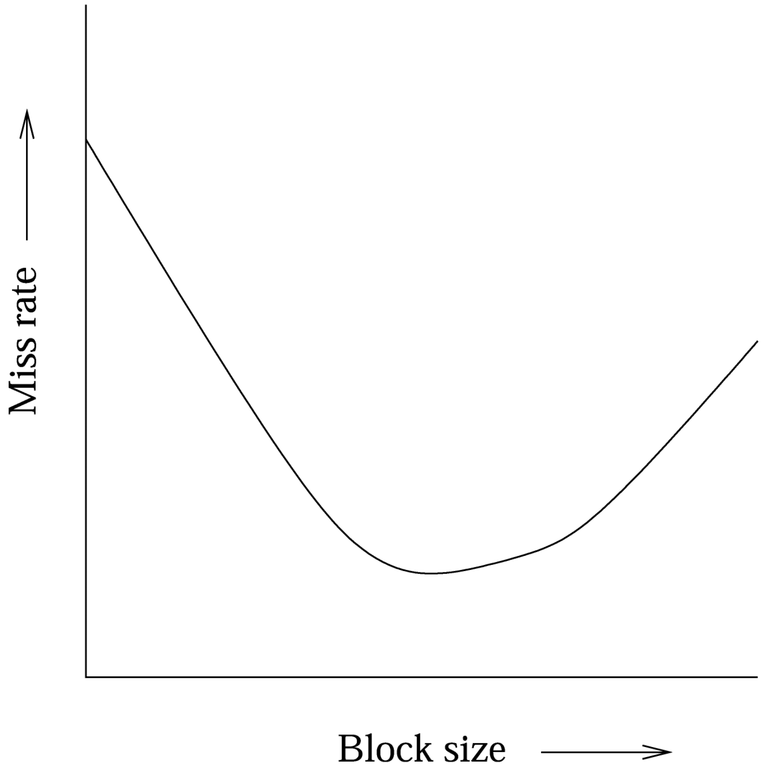
- Various policies used by cache
  - ∗ Placement of a block
    - » Direct mapping
    - » Fully associative mapping
    - » Set-associative mapping
  - ∗ Location of a block
    - » Depends on the placement policy
  - ∗ Replacement policy
    - » LRU is the most popular
      - – Pseudo-LRU is often implemented
  - ∗ Write policy
    - » Write-through
    - » Write-back

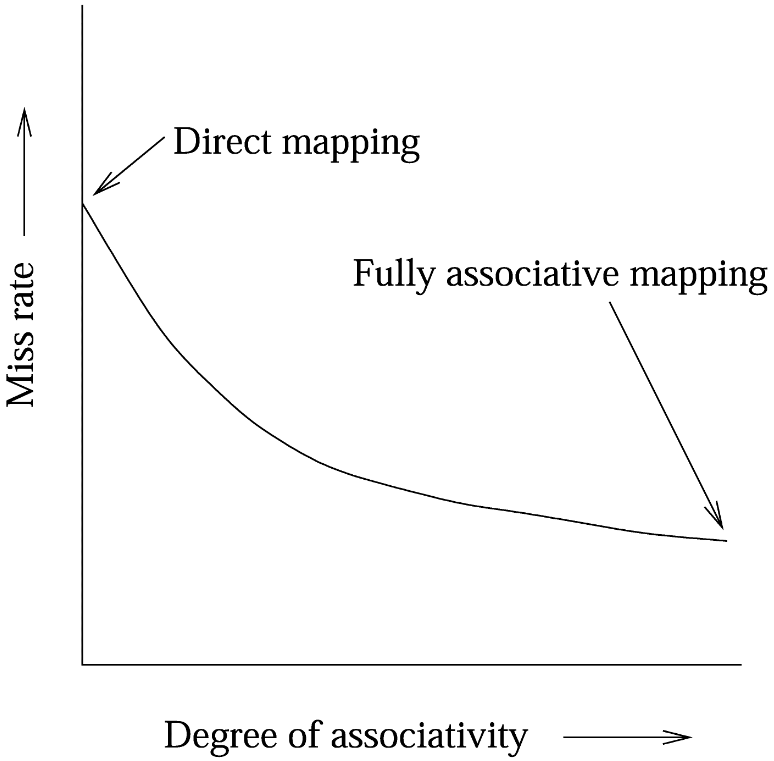
Design Issues

- Several design issues
  - ∗ Cache capacity
    - » Law of diminishing returns
  - ∗ Cache line size/block size
  - ∗ Degree of associativity
  - ∗ Unified/split
  - ∗ Single/two-level
  - ∗ Write-through/write-back
  - ∗ Logical/physical



Design Issues (cont’d)





Last slide