Santa Clara University

COEN 210

Chapter 5, part 2 – Cache

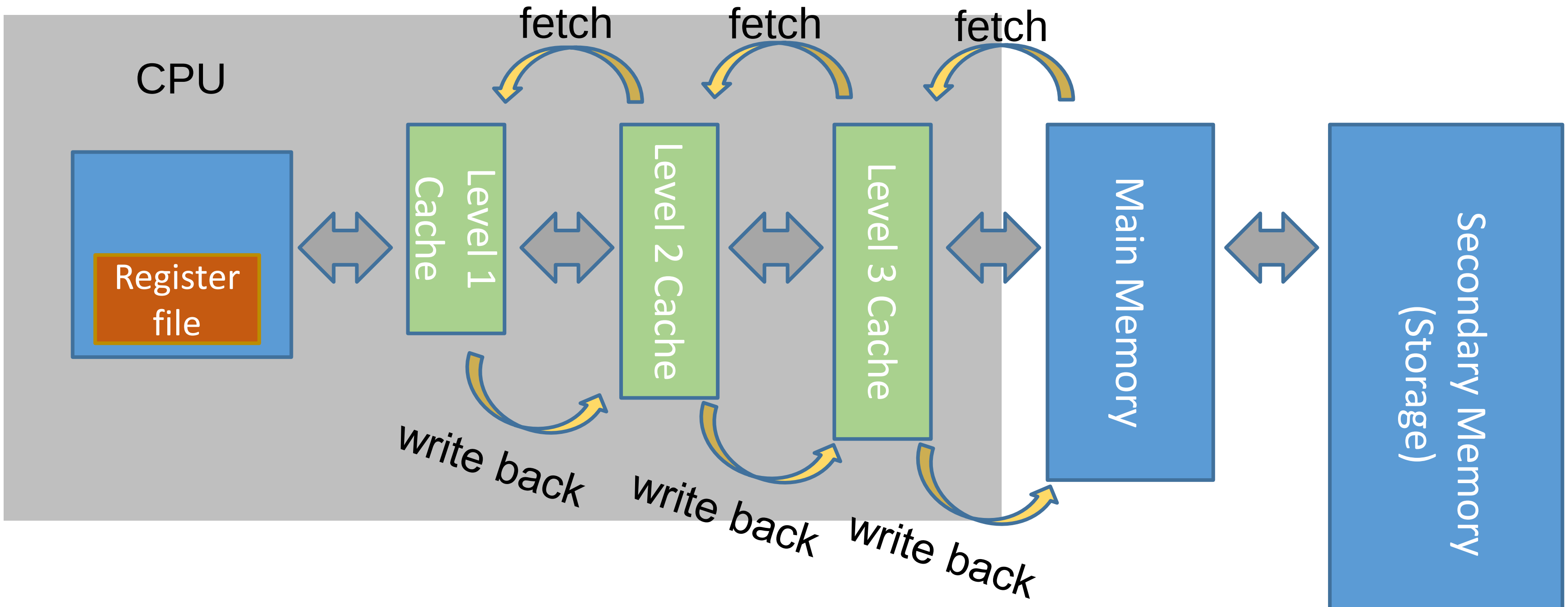
Yan Cui
ycui4@scu.edu

Where are we?

- Chapter 5 - memory hierarchy
  - Chapter 5.1 – Memory Technology and Organization
  - Chapter 5.2 - Caches
  - Chapter 5.3 - Virtual Memory

Cache Memory Performance

Cache Read/Write



- Process
  - For read data, it is checked for presence in each level
    - Data is fetched all the way up to L1 cache
    - Replacement may happen here
    - Data is then read by CPU on L1 cache
  - For written data, it is propagated back from L1 cache to main memory when it is replaced

Hit Or Miss

- Cache hit : data is present
  - Hit time : time to access data when it is present

- Cache miss : data is not present
  - Miss penalty : Extra delay caused by each cache miss

- Hit rate and miss rate
  - $hit\ rate = \frac{cache\ hits}{total\ access}$
  - $miss\ rate = 1 - hit\ rate$

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses

- So CPU time = (CPU execution clock cycles + Memory-stall clock cycles) * Clock cycle time

$$\begin{aligned}&\text{Memory stall cycles}\\&= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}\\&= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\end{aligned}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- How much faster a processor would run with a perfect cache that never missed?
- Solution:
- Miss cycles per instruction
  - I-cache: 0.02 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44

- So, Ideal CPU is 5.44/2 =2.72 times faster

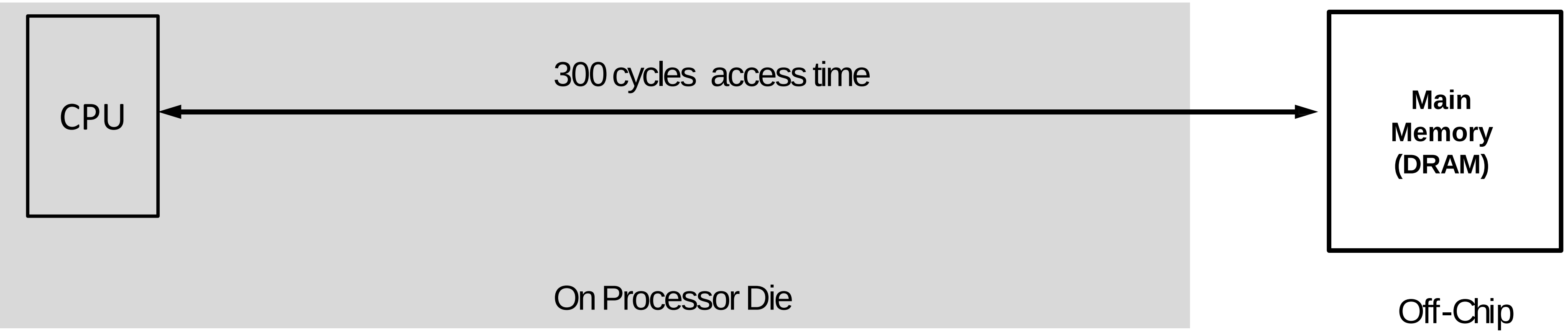
Average Memory Access Time

- To capture the fact that the time to access data for both hits and misses affects performance
- Average memory access time (AMAT) to examine alternative cache designs.
- AMAT: the average time to access memory considering both hits and misses and the frequency of different accesses.
  - AMAT = Hit time + Miss rate × Miss penalty

- Example
- Given:
  - CPU with 1ns clock cycle time, hit (cache access) time = 1 cycle, miss penalty = 20 cycles, miss rate = 5% per instruction
- Find the AMAT?
- Solution:
  - AMAT = Times for a hit + Miss rate * Miss penalty

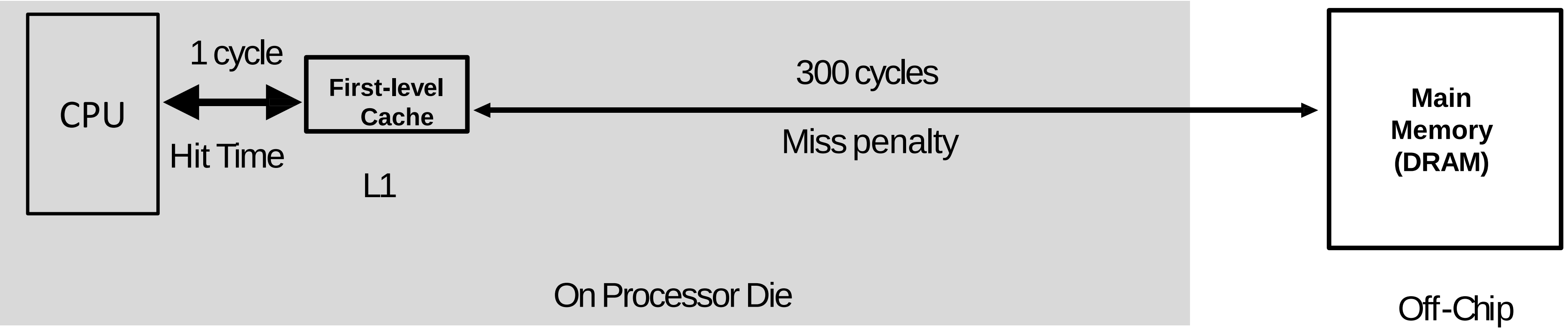
    = 1 + 0.05 × 20 = 2 clock cycles
- 2 cycles per instruction

AMAT without Cache



- AMAT is fixed, = 300 cycles
- CPU is pretty much idling all the time!!

AMAT with 1-Level Cache



- Average Memory Access Time (AMAT)

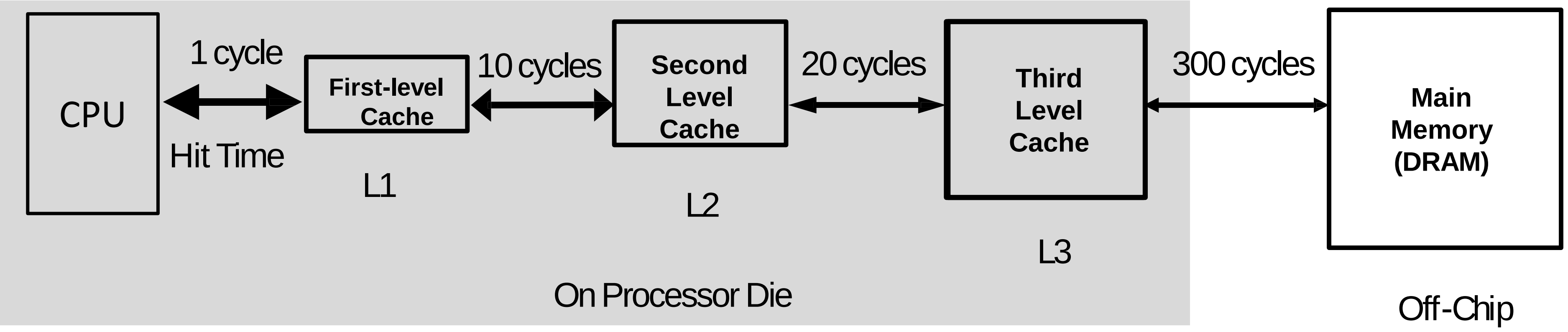
  = Hit Time + Miss rate x Miss Penalty

  = T_{hit}(L1) + Miss rate(L1) x T(Memory)

- For Example
  - Cache Hit Time = 1 cycle
  - Miss rate = 10% = 0.1
  - Miss penalty = 300 cycles
  - AMAT = 1 + 0.1 x 300 = 31 cycles

31 cycles is still very long.
Can we reduce it further?

Reducing Penalty with Multi-Level Cache



- AMAT for a multi-level cache organization

AMAT(L1)

$= T_{hit}(L1) + MissRate(L1) \times AMAT(L2)$

$= T_{hit}(L1) + Miss(L1) \times (T_{hit}(L2) + Miss(L2) * AMAT(L3))$

$= T_{hit}(L1) + Miss(L1) \times (T_{hit}(L2) + Miss(L2) * (T_{hit}(L3) + Miss(L3) * T(memory)))$

- Suppose miss rate for L1/L2/L3 are 10%, 5% and 1% respectively

AMAT(1) = 1 + 0.1 x [10 + 0.05 x { 20 + 0.01 x 300 }] = 2.115 cycles

Acceptable speed

Cost of Multi-Level Cache

- Assume we have following multi-level cache system
  - L1 cache : 16KB
  - L2 cache : 128KB
  - L3 cache : 1M
  - Main memory : 1GB

- Assume the following pricing
  - L1 cache : $6000/GB
  - L2 cache : $4000/GB
  - L3 cache : $2000/GB
  - Main memory : $20/GB

- Total cost for the memory system is $6000 \ * \frac{16KB}{1GB} + 4000 \ * \frac{128KB}{1GB} + 2000 * \frac{1MB}{1GB} + 20 =$ $22.53

Comparison of Hypothetical 1GB Memory System

Types	Cost (USD)	AMAT (cycles)
Only L1 cache	$6000	1
Only L2 cache	$4000	10
Only L3 cache	$2000	20
Only RAM	$20	300
Multi-level cache + RAM	$22.53	2.115

There Is Only One Problem …

- Why can a 16KB L1 cache achieve 90% or higher hit rate?
- If purely random access
  - $hit\ rate = \frac{16KB}{1\ GB} \cong 0.0015\%$
- Typical real world total cache hit rate is around 95%
- How to increase hit rate?
  - Larger cache size
  - Improve software memory access locality

Cache Organization

Cache Design Considerations

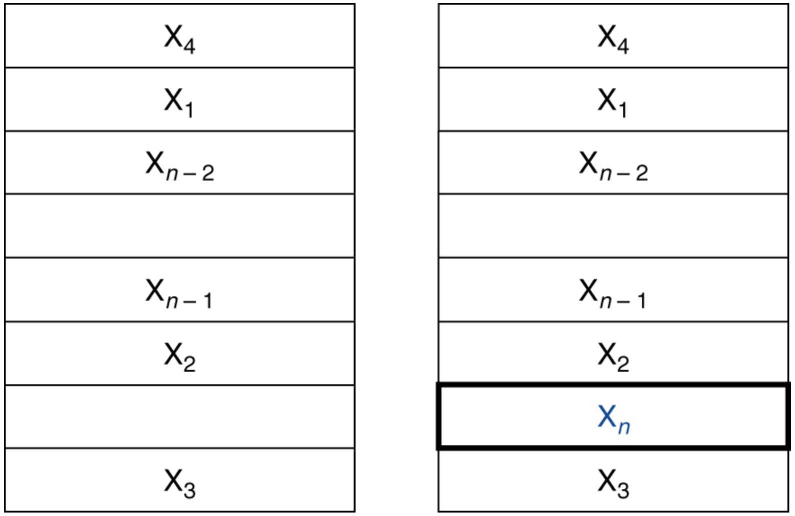
- Cache geometry
  - Cache block size and number of blocks
- Cache line structure
  - Valid bit for presence
  - Address tagging for address matching
- Mapping method
  - Which data entry is stored in a cache line?
  - Because main memory is larger than cache size
- Replacement strategy
  - What happens when cache is full?
- Write strategy
  - What to do when an access is a write operation?

One objective : explore data locality and improve hit rate

§5.3 The Basics of Caches

Cache Memory

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

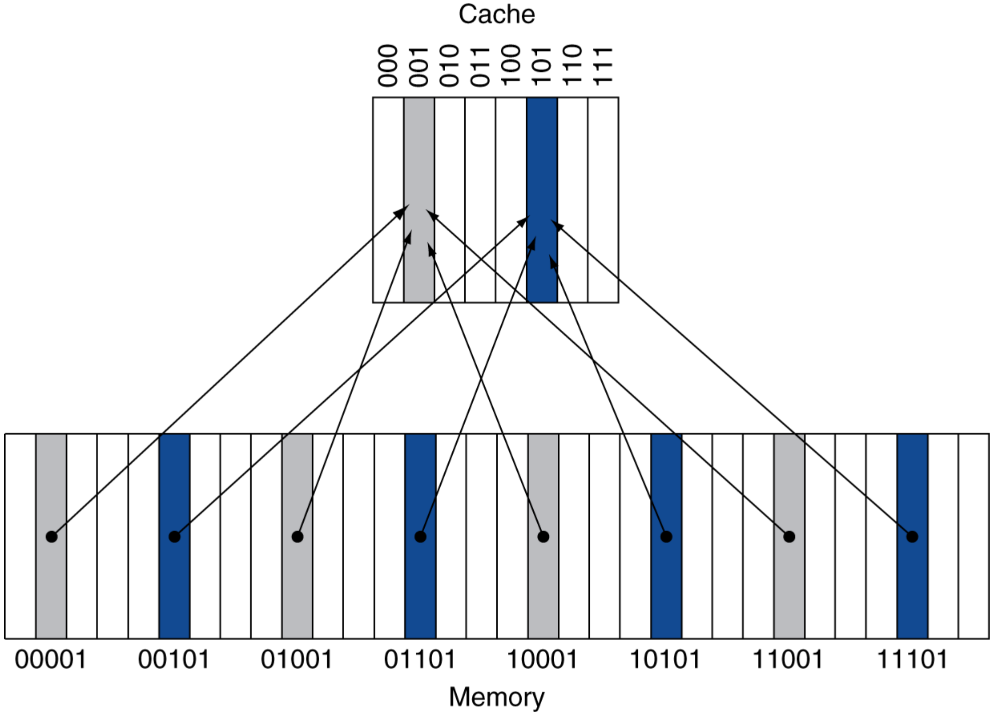


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present (in the cache)?
- If it is, how do we find it?

Direct Mapped Cache

- Cache location determined by memory address
- Direct mapped: only one choice
  - Each memory location is mapped directly to exactly one location in the cache.
  - (Block address) modulo (#Blocks in cache)



A direct-mapped cache with eight entries showing the addresses of memory words between 0 to 31 that map to the same cache locations

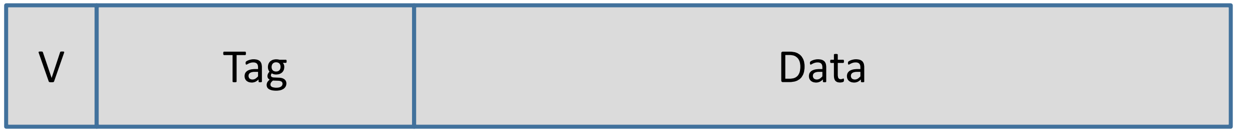
- If # of Blocks is a power of 2 in the cache
- Use low-order $\log_2$(#blocks in cache) bits of the address
- E.g. 8 words in the cache
  - an address X maps to the direct-mapping cache word Xmodulo 8.
  - The lower order of log2(8) =3 bits are used as the cache index
  - Thus, addresses 00001, 01001, 100001, and 11001 all map to entry 001 of the cache

Tags and Valid Bits

- Problem:
  - each cache location can contain the contents of a number of different memory locations
- How do we know which particular memory block is stored in a cache location?
  - Store address information as well as the data
  - Actually, only need the high-order bits, corresponding to the bits that are not used as an index into the cache
    - E.g., In previous figure, have the upper two of the five address bits
  - Called the tag
- What if there is no data in a cache location?
  - Valid bit: 1/ Y= present, 0 / N = not present
  - Initially 0

Cache Line (Cache Block) Structure

Cache Line == Cache Block, will be used interchangeably.



- V – valid bit
- Tag – indicate which memory address is the data from currently
- Data – Actual bits stored
  - The data size in cache line is called cache line size or cache block size
  - In practice, cache line size can range 16 bytes to 128 bytes

Cache Example

- A sequence of nine memory reference to an empty eight-block cache
- 8-blocks, 1 word/block, direct mapped

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

The initial state of the cache after power-on

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

After handling a miss of address 10110

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

After handling a miss of address (11010)

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision

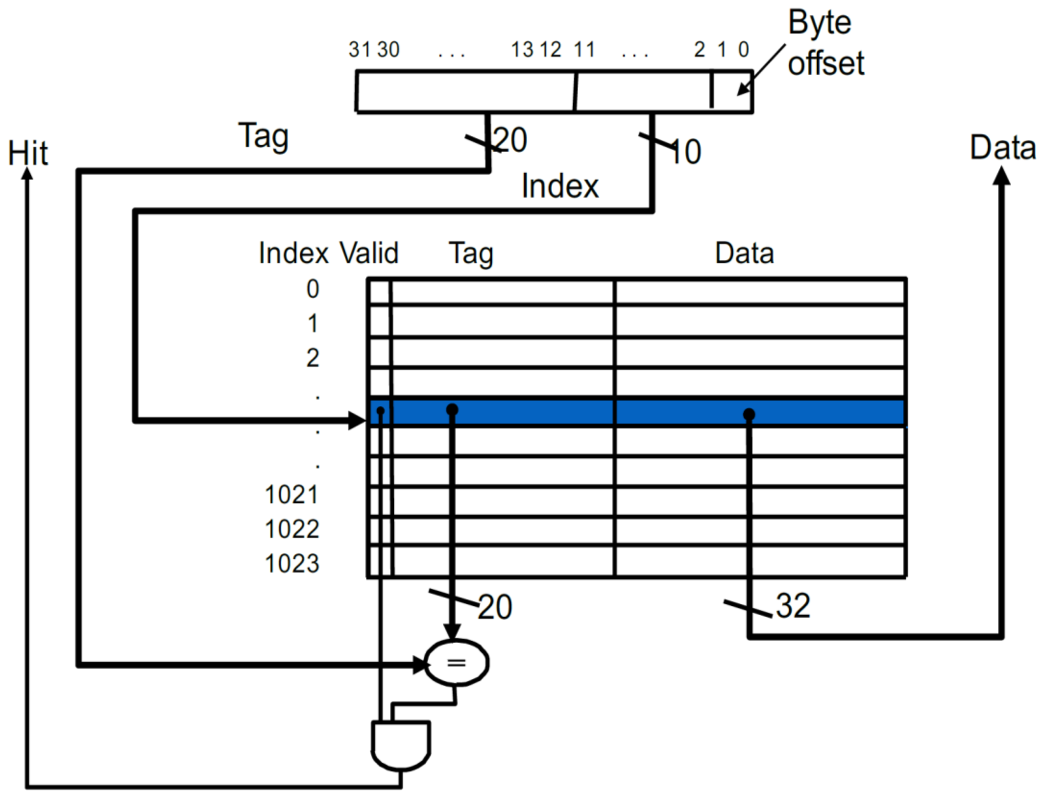
A reference address is divided into:

- A tag field
- A cache index

Words are aligned to multiple of 4 bytes

Byte offset: The least significant two bits specify a byte within a word

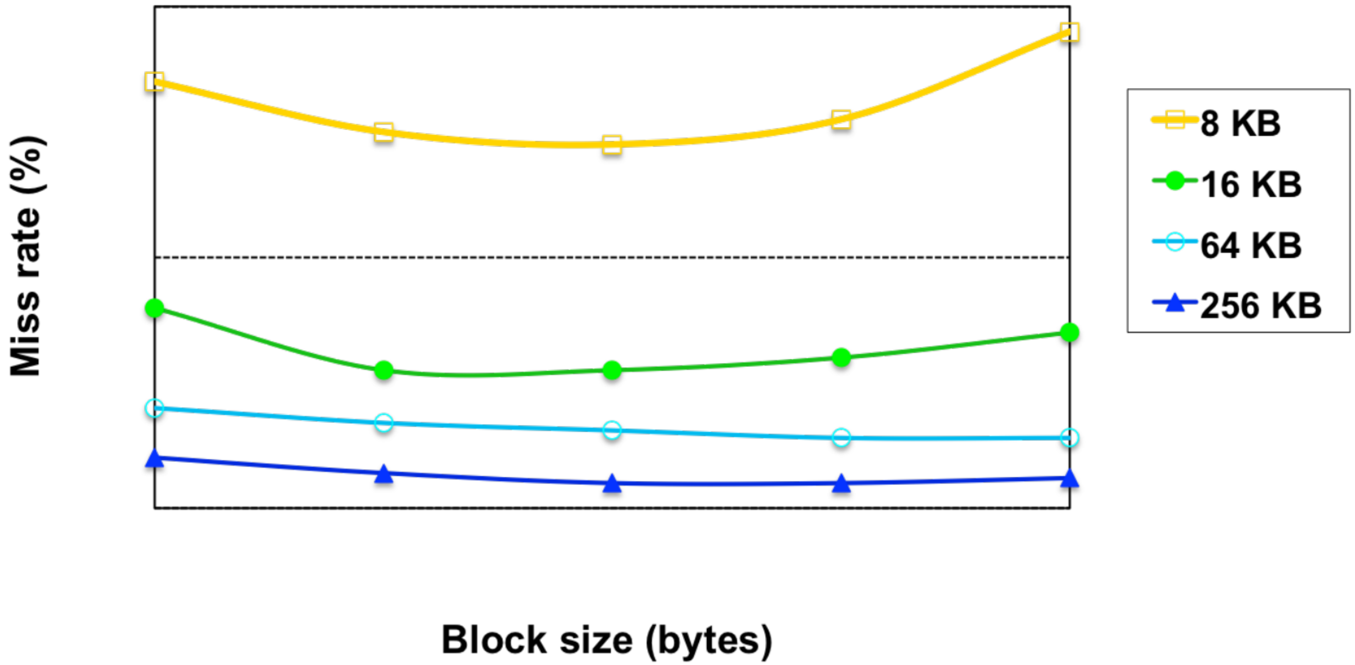
- Ignored when selecting a word in the block



Example: Mapping an Address to a Multiword Cache Block

- Given:
  - 64 blocks, 16 bytes/block
- To what block number does byte address 1200 map?
- Solution:
  - (Block address) modulo (Number of blocks in the cache)
  - Block address =byte address / bytes per block = $\lfloor 1200/16 \rfloor$ = 75
  - Block number = 75 modulo 64 = 11
- In fact, this block maps all address between 1200 and 1215

Miss Rate v.s. Block Size v.s. Cache Size



Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Increasing block size => increasing the miss penalty
    - Two parts: the latency to the first word + the transfer time for the rest of the block
  - Can override benefit of reduced miss rate
  - Solution: Early restart and critical-word-first can help

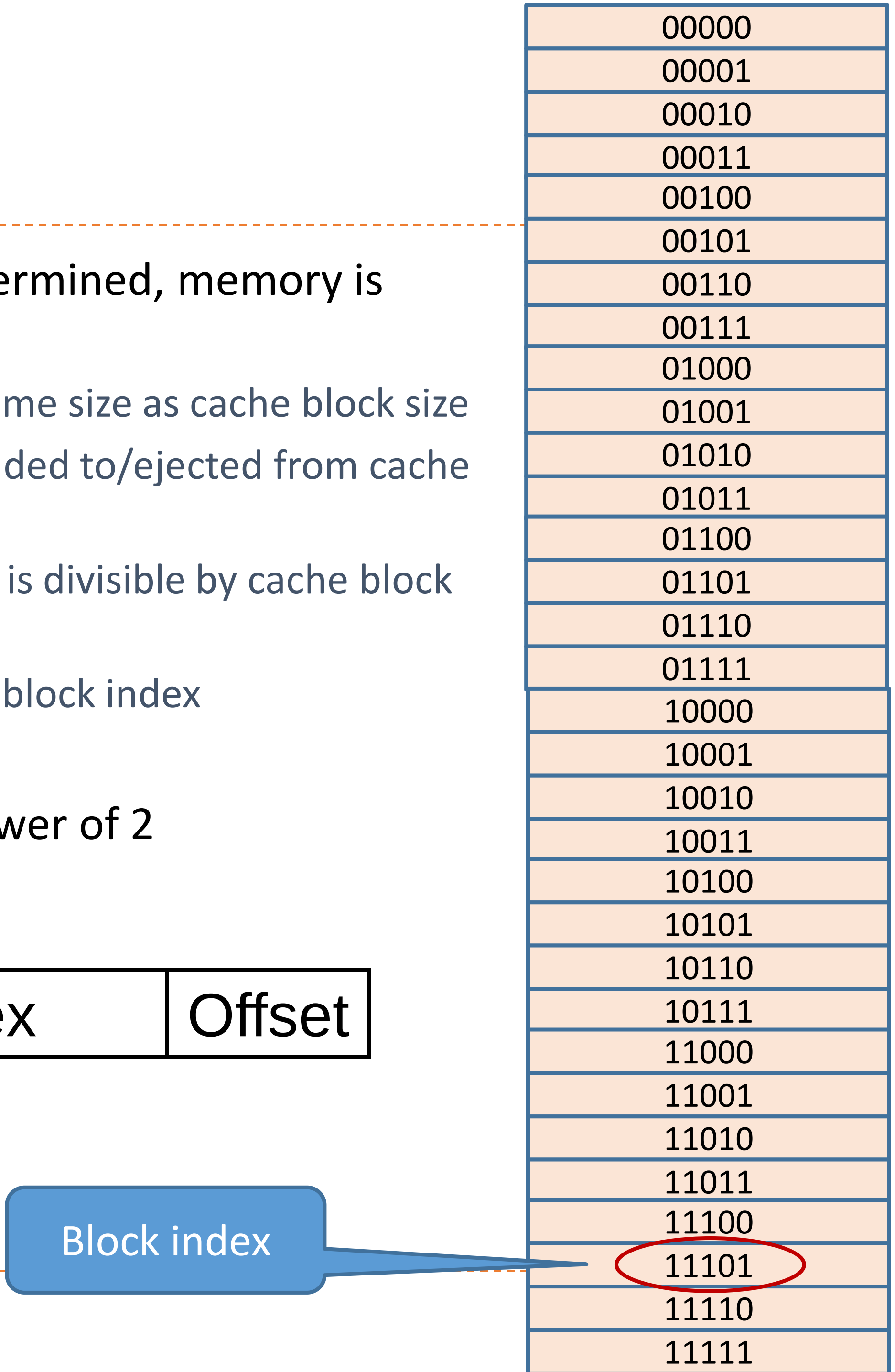
- Typical sizes are 4, 8, 16 words, or 16, 32, 64 bytes
  - Always a power of 2!

Memory Blocks

- When cache block size is determined, memory is divided into mapping blocks
  - Each memory block has the same size as cache block size
  - Each memory block will be loaded to/ejected from cache as a whole
  - Starting address of each block is divisible by cache block size
  - Each block is identified by the block index

- Since cache block size is a power of 2

Memory address	Block Index	Offset



00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

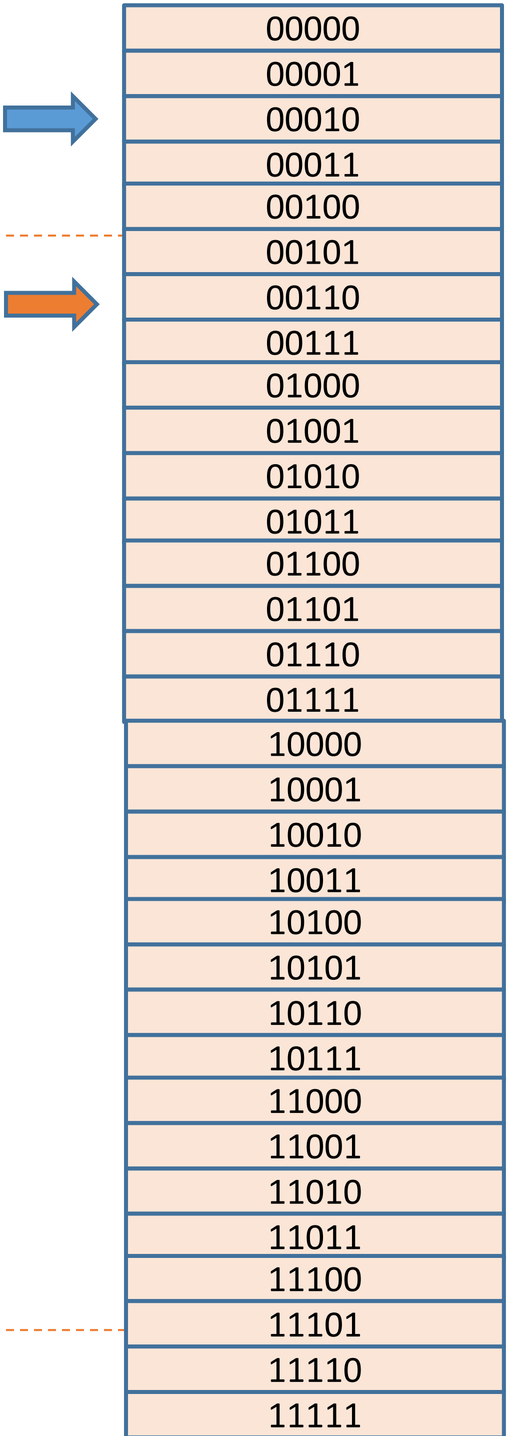
Memory Block Example

- Assume memory block size is 16 byte
  - We need at least 4 bits to represent offset
- Assume we have 32 blocks
  - We need at least 5 bits to presents block index
- What is total memory size?
  - 16 x 32 = 512 bytes
- What is the address range for 3rd block or block 00010_2?
  - 32_{10} ~ 47_{10}

Block Index | Offset

Block Index | Offset

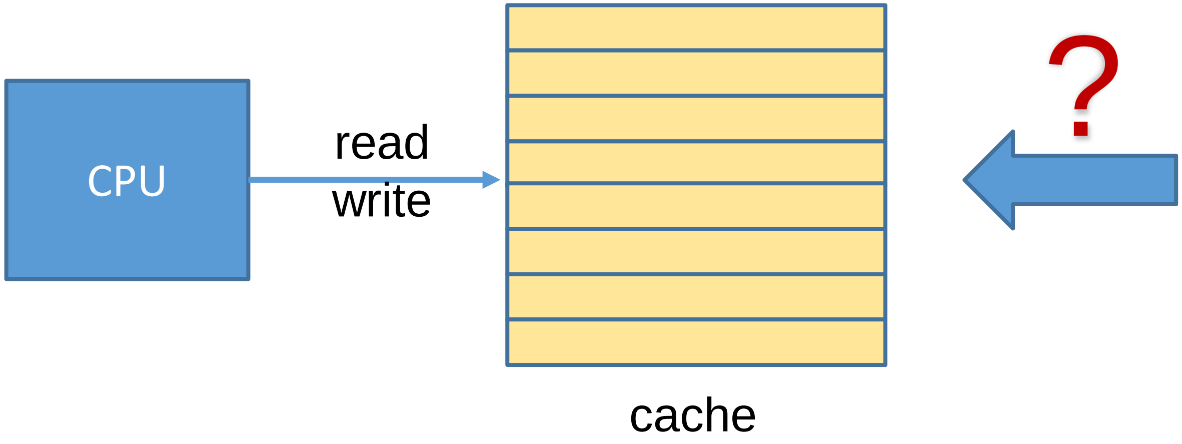
  - Memory address range: $00010\ 0000_2$ ~ $00010\ 1111_2$
- Which block does memory byte at address 100_{10} fall into?
  - $100_{10} = 00110\ 0100_2$
  - Memory block index is 00110_2 or 6_{10}



00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Mapping

Mapping determine which memory block gets loaded into which cache block.



00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Goals:

- Low complexity/resource
- Reduce miss rate and CPU access time

Mapping Methods

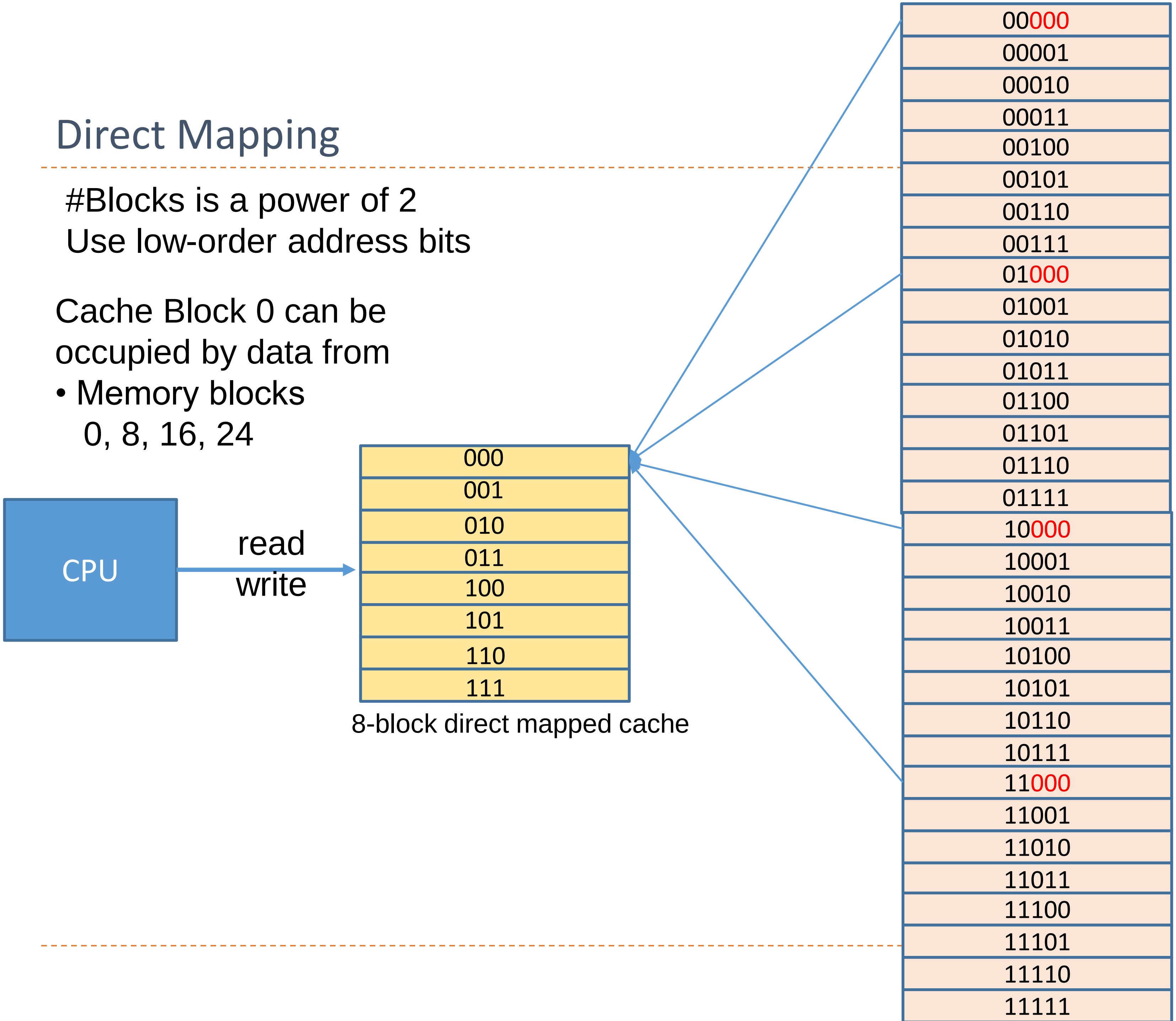
- Direct-mapped: each a memory block can be placed in exactly one location in the cache.
  - Some of the memory address bit are used to index the cache
- N-way set-associative: each memory block can be placed into n locations
- Fully-associative: a memory block can be placed in any location in the cache
  - Cache location and memory address are unrelated

Direct Mapping

#Blocks is a power of 2
Use low-order address bits

Cache Block 0 can be occupied by data from

- Memory blocks
  0, 8, 16, 24



000
001
010
011
100
101
110
111

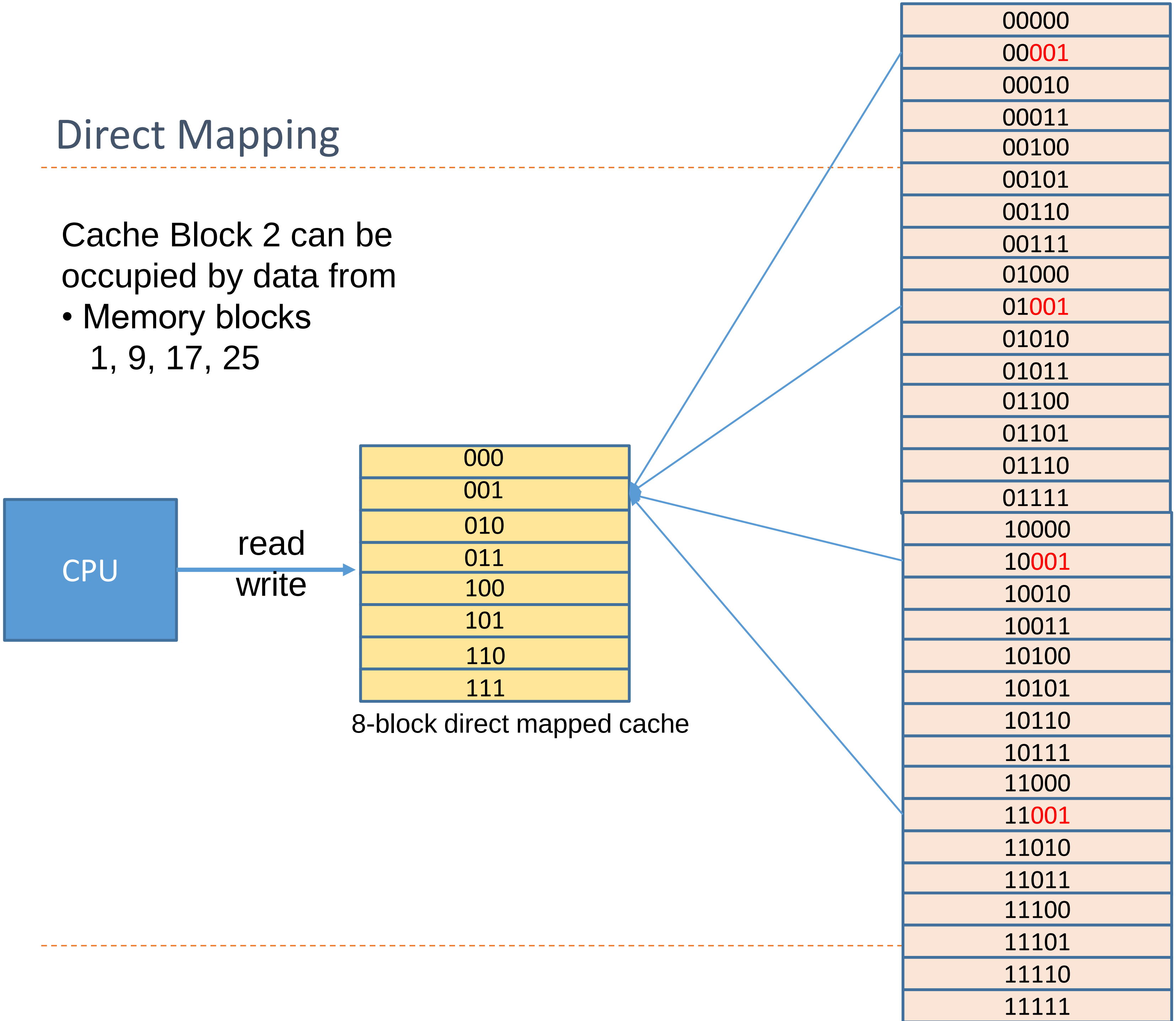
8-block direct mapped cache

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping

Cache Block 2 can be occupied by data from

- Memory blocks
  1, 9, 17, 25



CPU

read
write

000
001
010
011
100
101
110
111

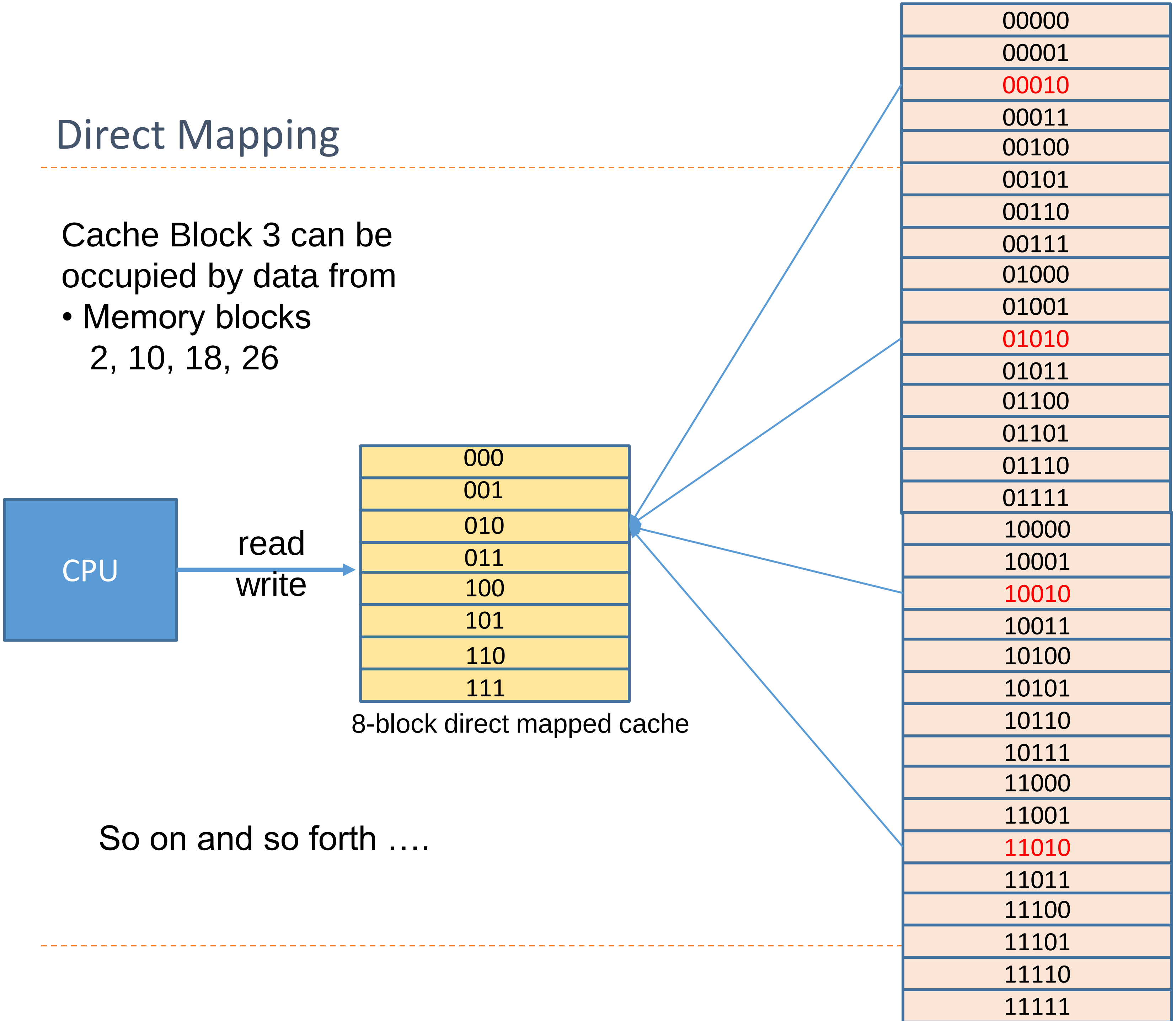
8-block direct mapped cache

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping

Cache Block 3 can be occupied by data from

- Memory blocks 2, 10, 18, 26



8-block direct mapped cache

So on and so forth ….

Summary: Direct Mapping

- Memory value can only be placed at a single corresponding location in the cache

- Fast indexing mechanism
  - Fast read/write from CPU
  - Cache index = memory block index % number of cache blocks
  - Assume number of cache blocks is 2^n (n = 3)

Cache line index = last n bits of memory block index

- Simple replacement:
  - Kick off the existing tenant

Accessing data in a direct mapped cache

Three types of events:

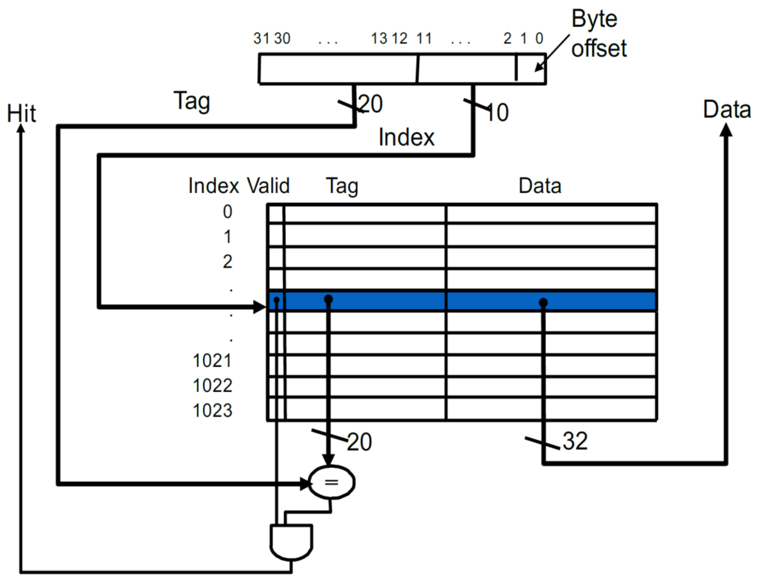
- cache hit: cache block is valid and contains proper address, so read desired word
- cache miss: nothing in cache in appropriate block, so fetch from memory
- cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory

Cache Access Procedure:

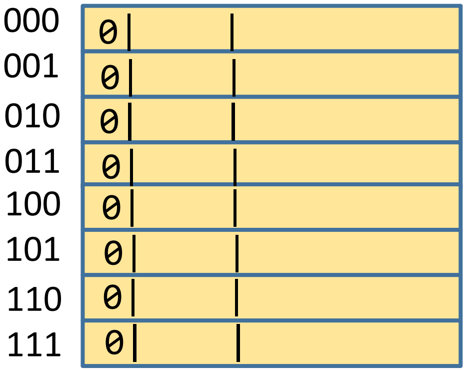
(1) Use Index bits to select cache block

(2) If valid bit is 1, compare the tag bits of the address with the cache block tag bit

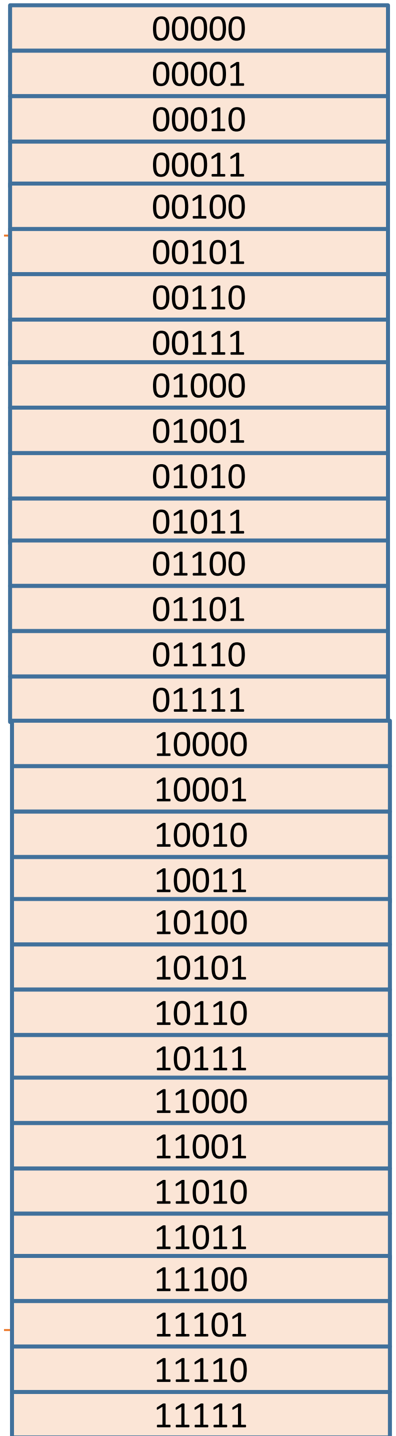
(1) If they match, use the offset to read out the word/byte



Direct Mapping Example



000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

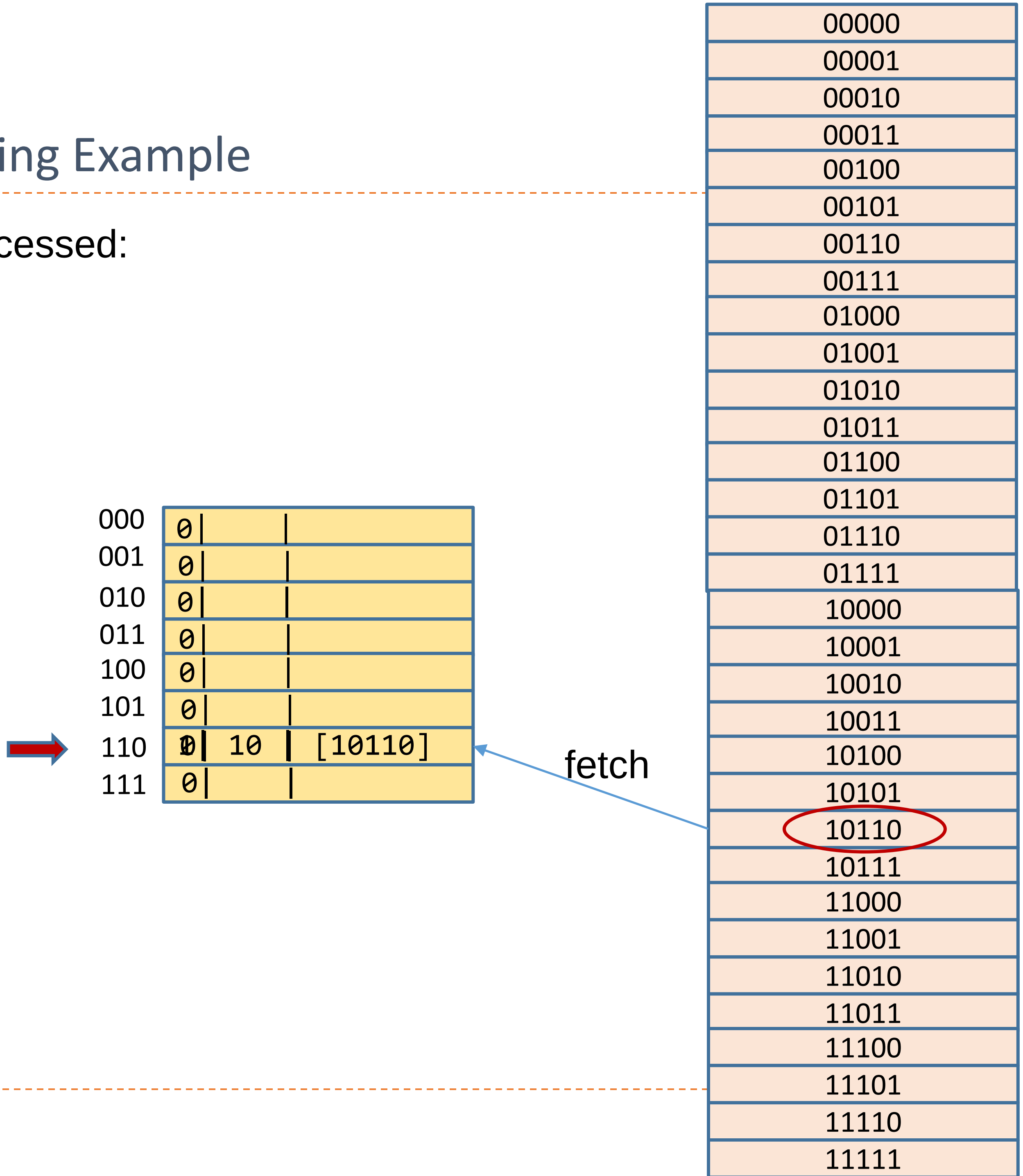


00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping Example

Address block accessed:

- 10 110



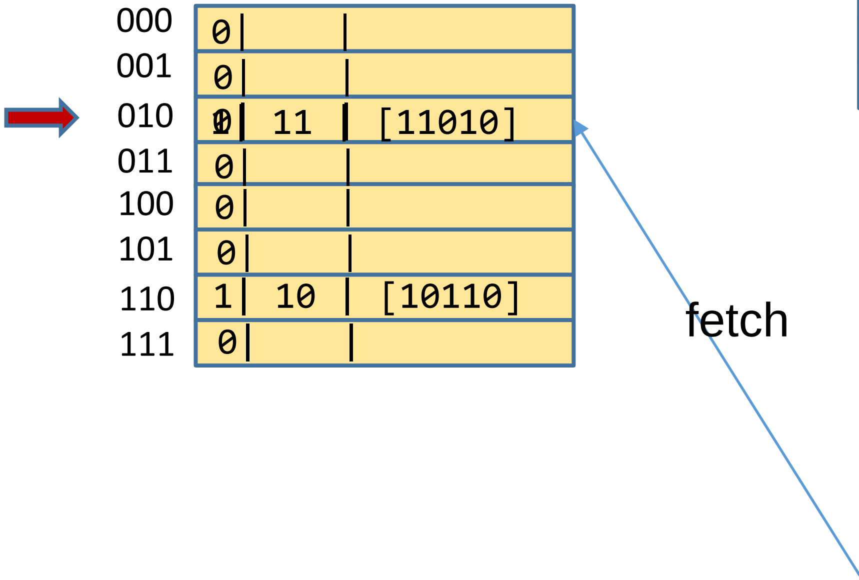
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	[10110]
111	0		

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

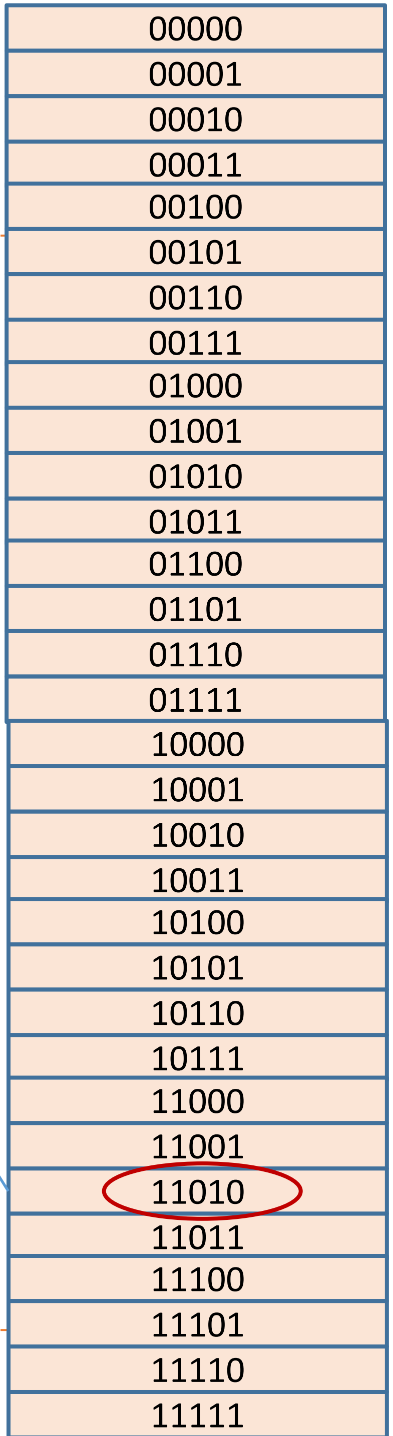
Direct Mapping Example

Address block accessed:

- 10 110
- 11 010



000	0		
001	0		
010	1	11	[11010]
011	0		
100	0		
101	0		
110	1	10	[10110]
111	0		

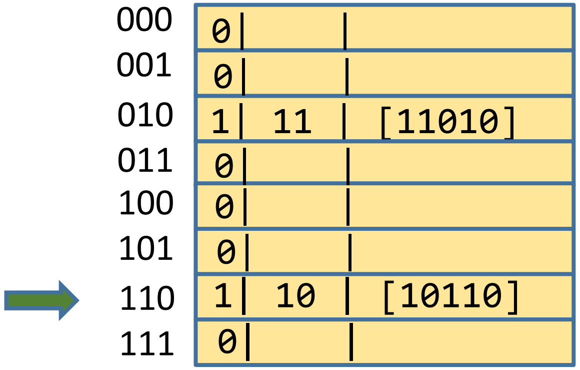


00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

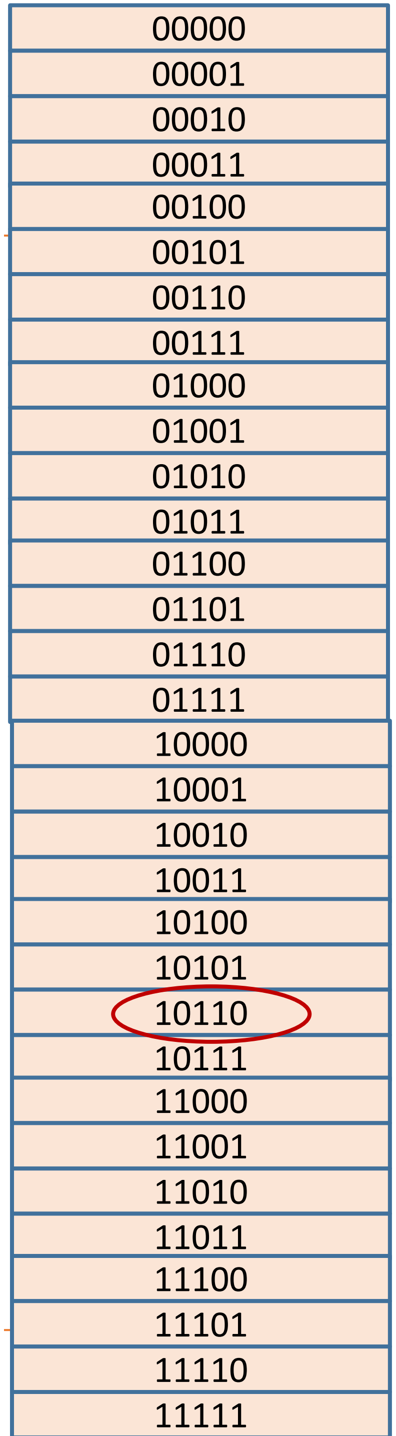
Direct Mapping Example

Address block accessed:

- 10 110
- 11 010
- 10 110



000	0		
001	0		
010	1	11	[11010]
011	0		
100	0		
101	0		
110	1	10	[10110]
111	0		

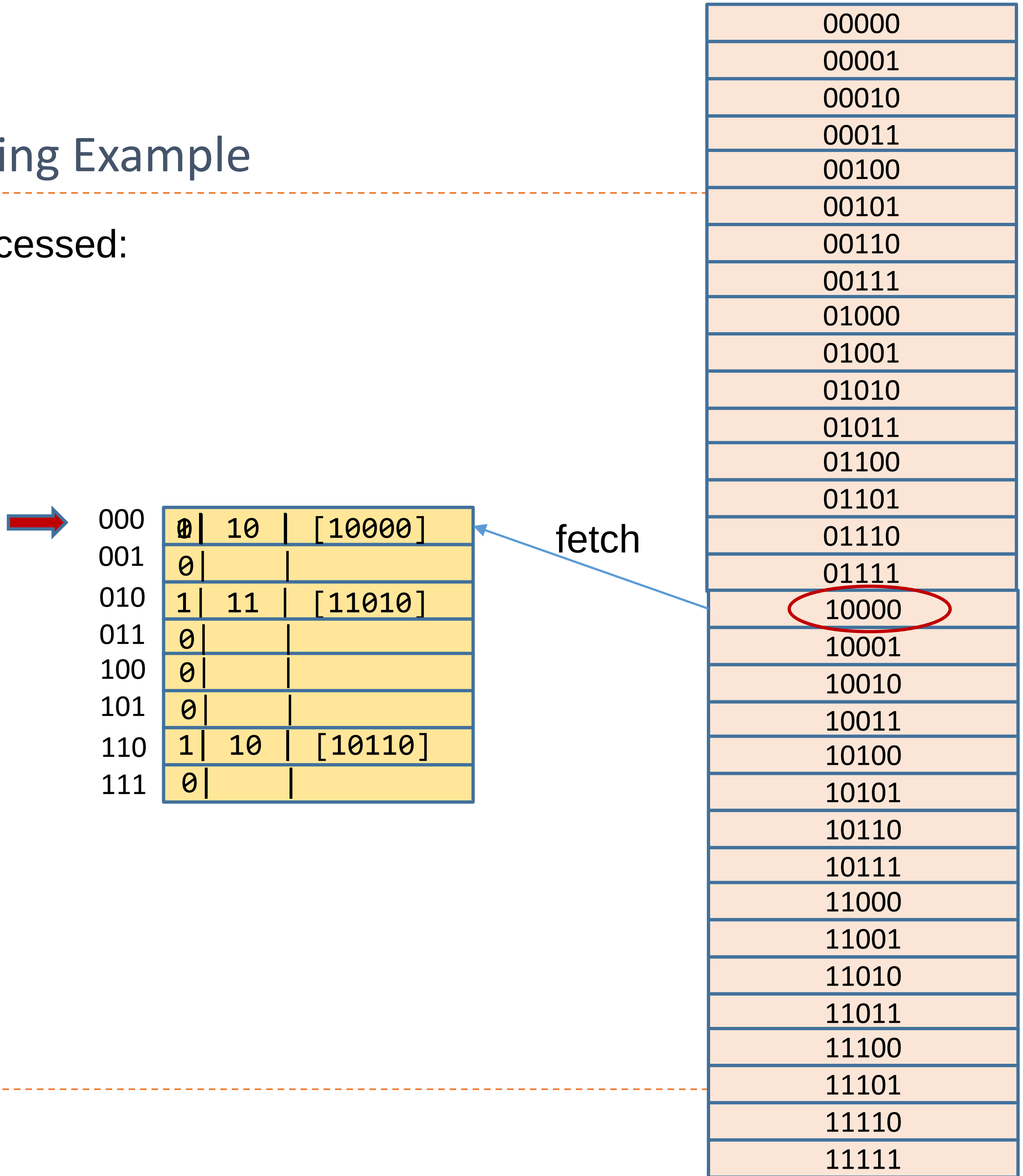


00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping Example

Address block accessed:

- 10 110
- 11 010
- 10 110
- 10 000



000	1	10	[10000]
001	0		
010	1	11	[11010]
011	0		
100	0		
101	0		
110	1	10	[10110]
111	0		

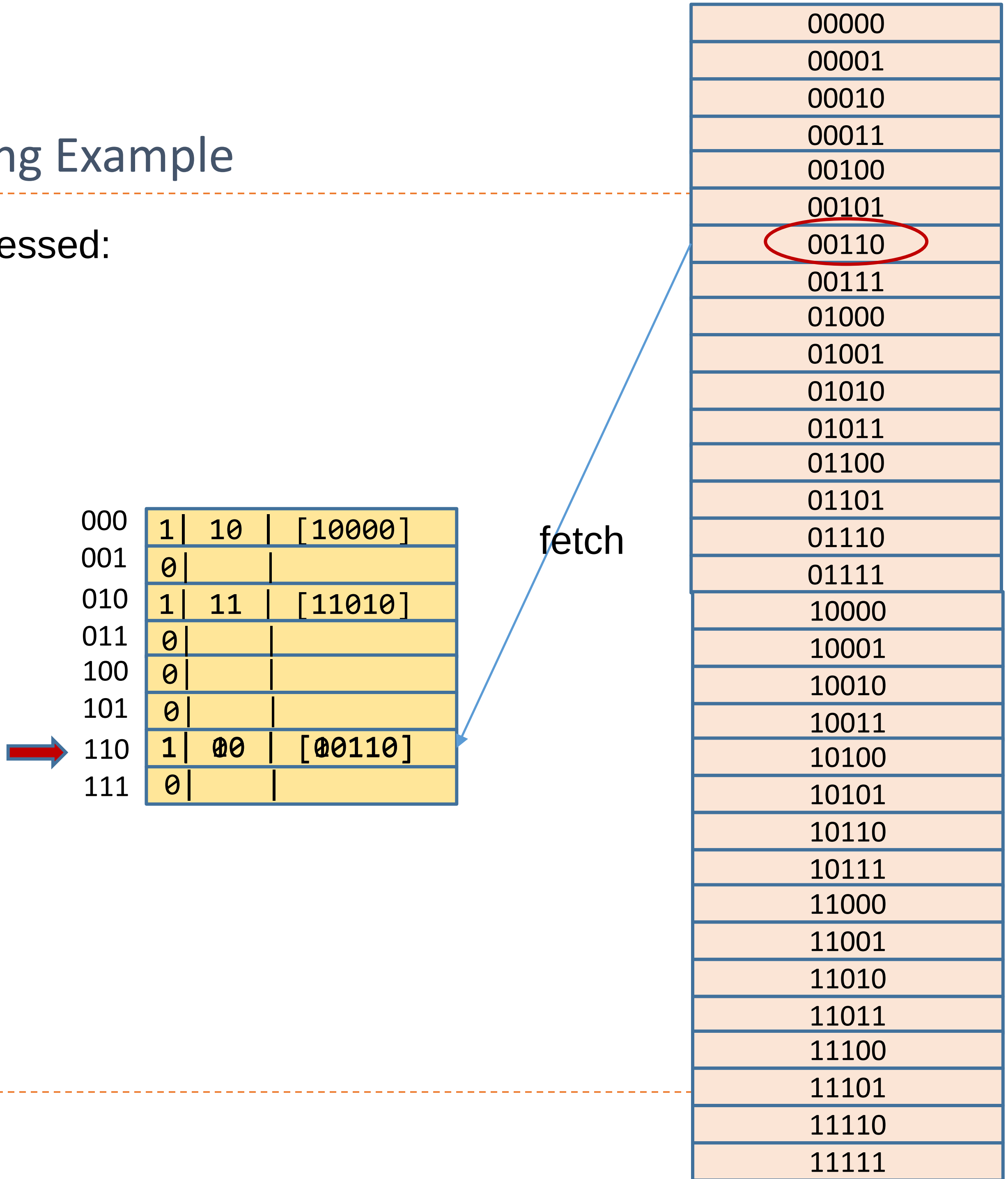
fetch

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping Example

Address block accessed:

- 10 110
- 11 010
- 10 110
- 10 000
- 00 110



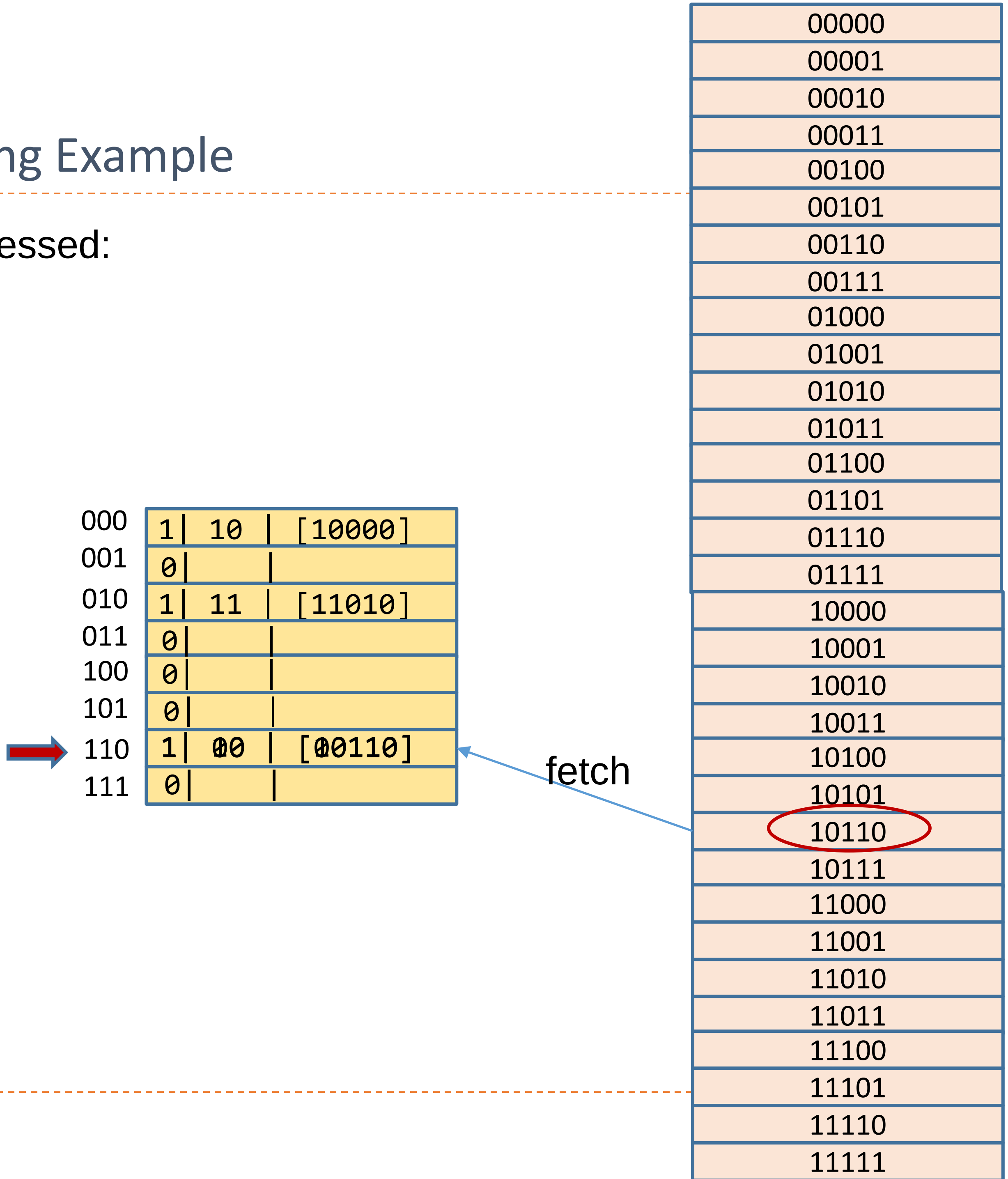
000	1	10	[10000]
001	0		
010	1	11	[11010]
011	0		
100	0		
101	0		
110	1	00	[00110]
111	0		

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping Example

Address block accessed:

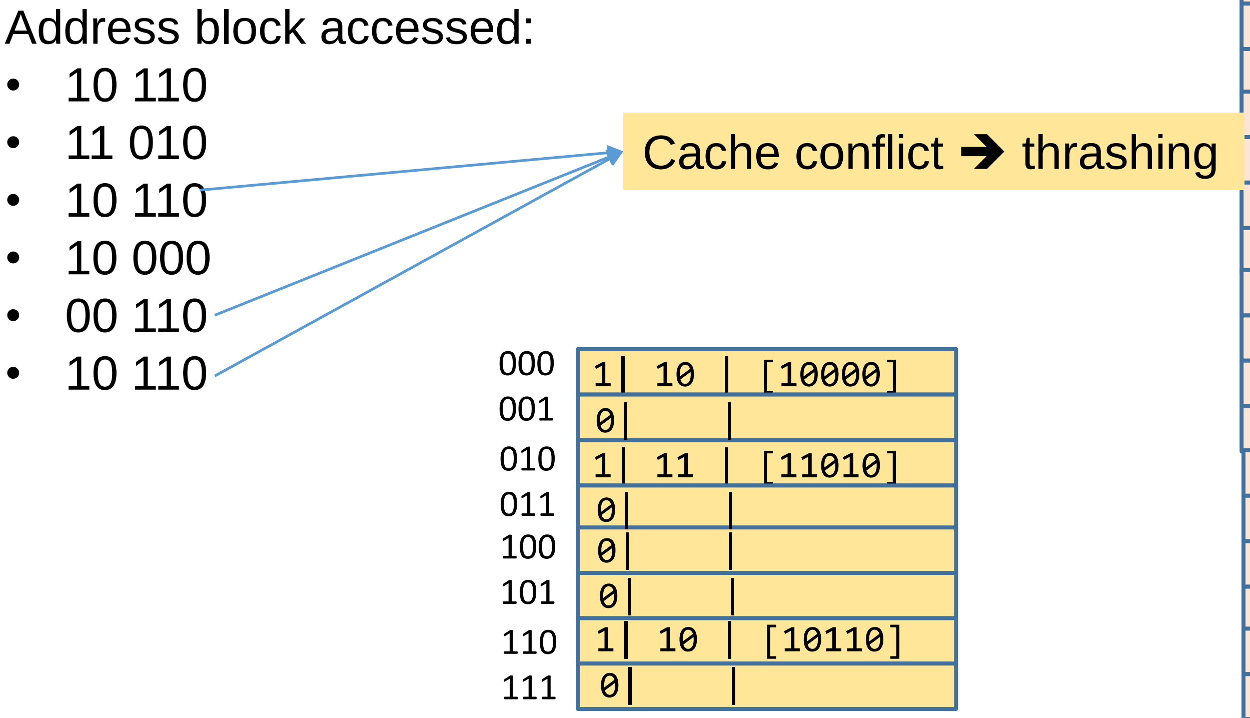
- 10 110
- 11 010
- 10 110
- 10 000
- 00 110
- 10 110



000	1	10	[10000]
001	0		
010	1	11	[11010]
011	0		
100	0		
101	0		
110	1	10	[10110]
111	0		

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Direct Mapping Example



Address block accessed:

- 10 110
- 11 010
- 10 110
- 10 000
- 00 110
- 10 110

Cache conflict ➔ thrashing

000	1	10	[10000]
001	0		
010	1	11	[11010]
011	0		
100	0		
101	0		
110	1	10	[10110]
111	0		

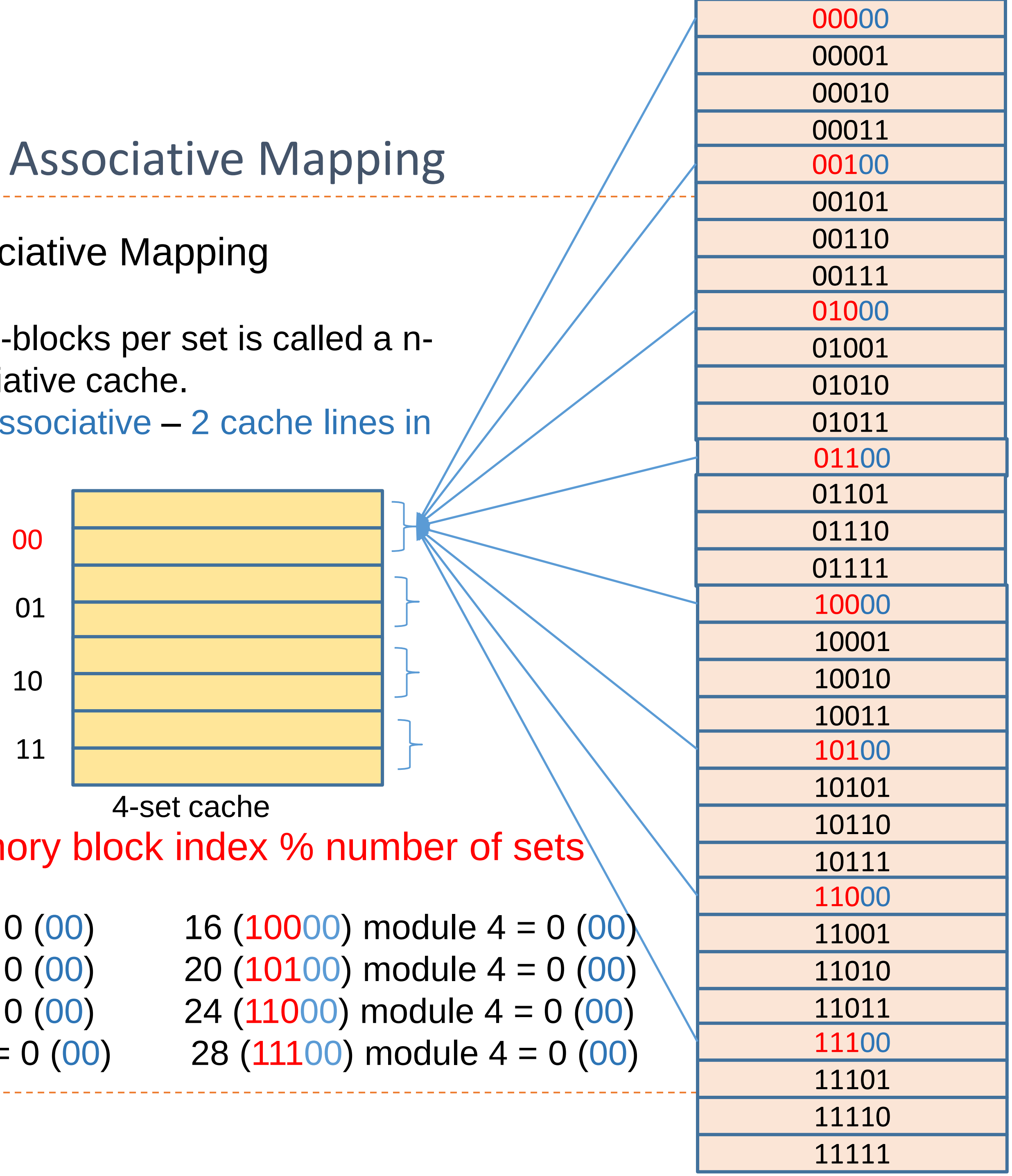
Conflict: when too many blocks try to load into the same set, some blocks will be replaced and later retrieved

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Two-Way Set Associative Mapping

n-way Set Associative Mapping

- A cache with n-blocks per set is called a n-way set associative cache.
- Two-way set associative – 2 cache lines in each set

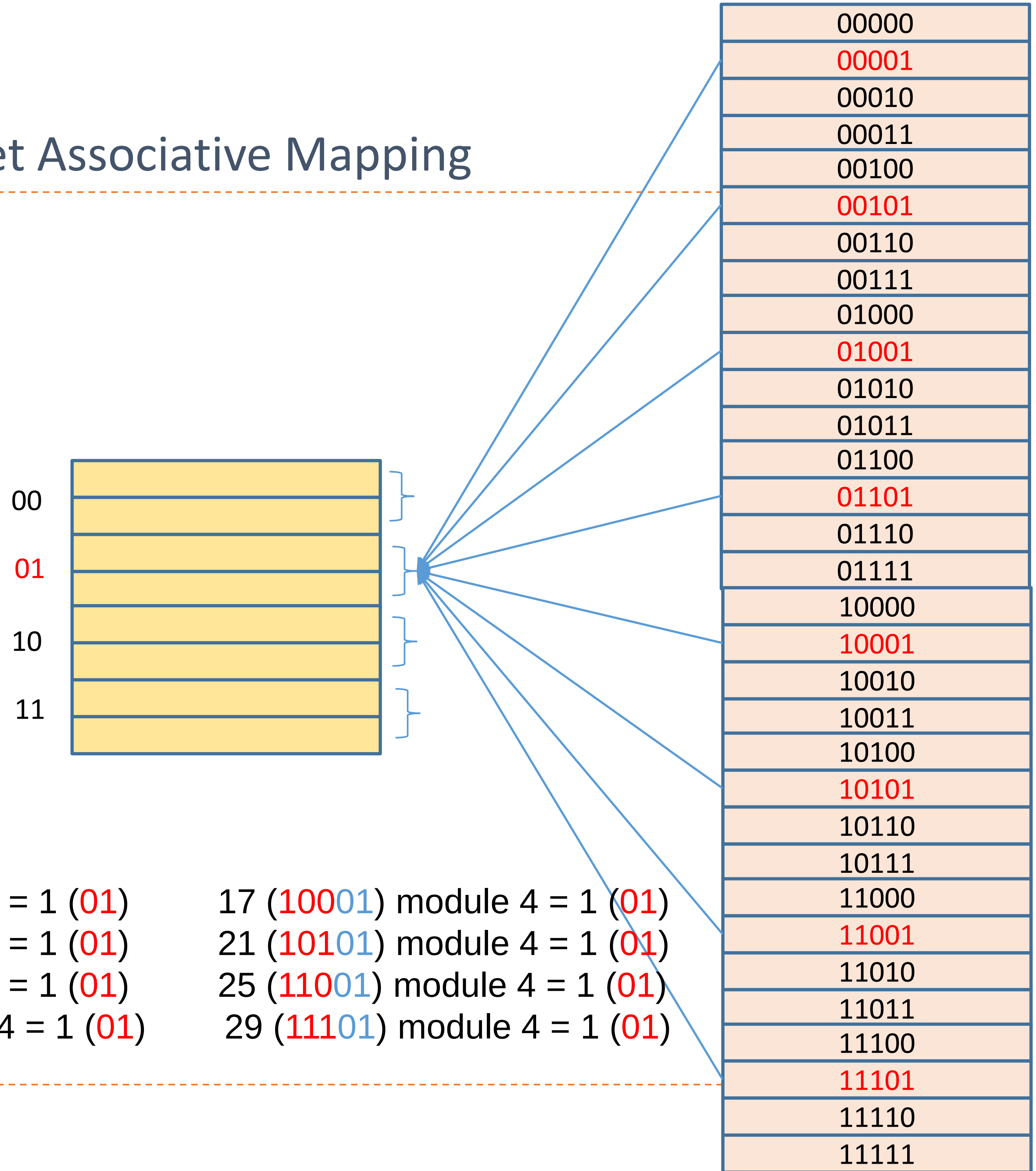


Set index = memory block index % number of sets

0 (00000) modulo 4 = 0 (00)
4 (00100) modulo 4 = 0 (00)
8 (01000) modulo 4 = 0 (00)
12 (01100) modulo 4 = 0 (00)

16 (10000) module 4 = 0 (00)
20 (10100) module 4 = 0 (00)
24 (11000) module 4 = 0 (00)
28 (11100) module 4 = 0 (00)

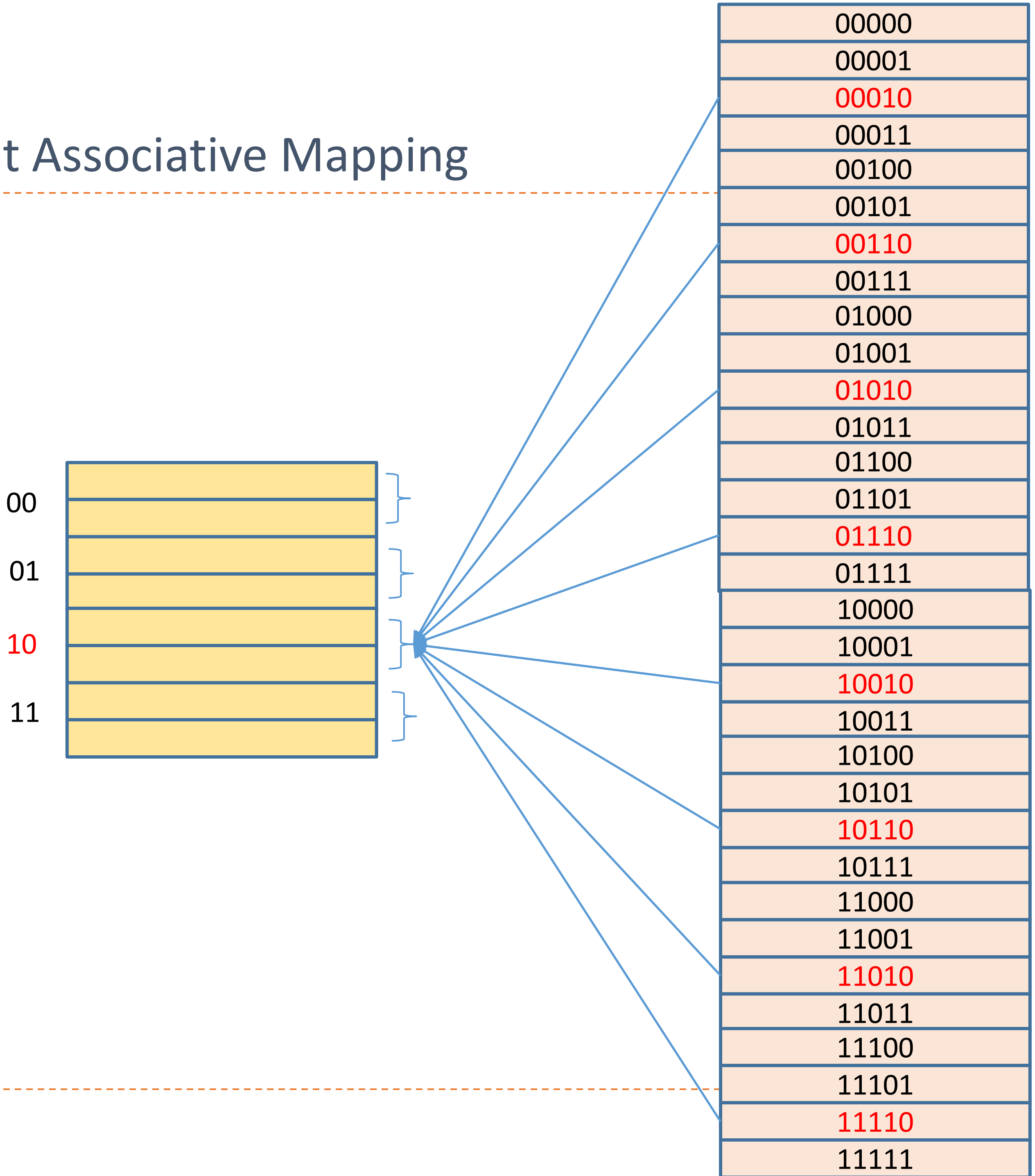
Two-Way Set Associative Mapping



1 (00001) modulo 4 = 1 (01)
5 (00101) modulo 4 = 1 (01)
9 (01001) modulo 4 = 1 (01)
13 (01101) modulo 4 = 1 (01)

17 (10001) module 4 = 1 (01)
21 (10101) module 4 = 1 (01)
25 (11001) module 4 = 1 (01)
29 (11101) module 4 = 1 (01)

Two-Way Set Associative Mapping



00

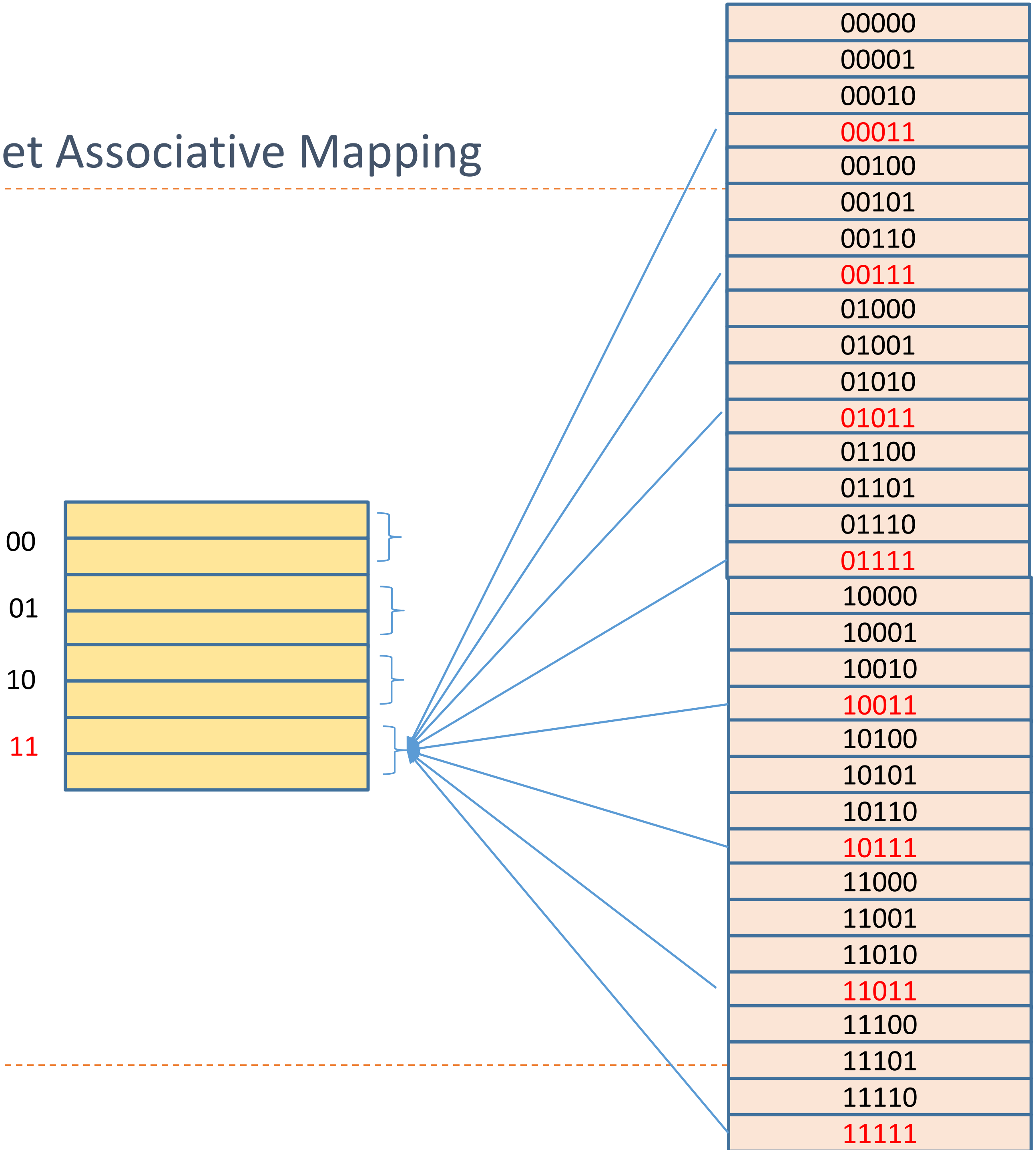
01

10

11

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

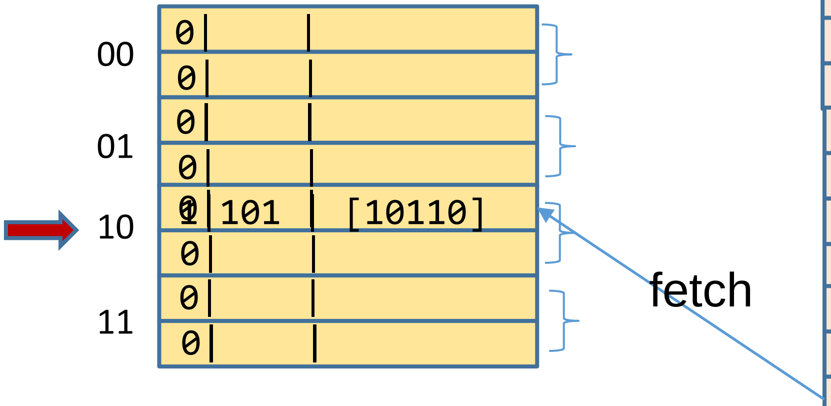
Two-Way Set Associative Mapping



Two-Way Set Associative Mapping Example

Address block accessed:

- 101 10

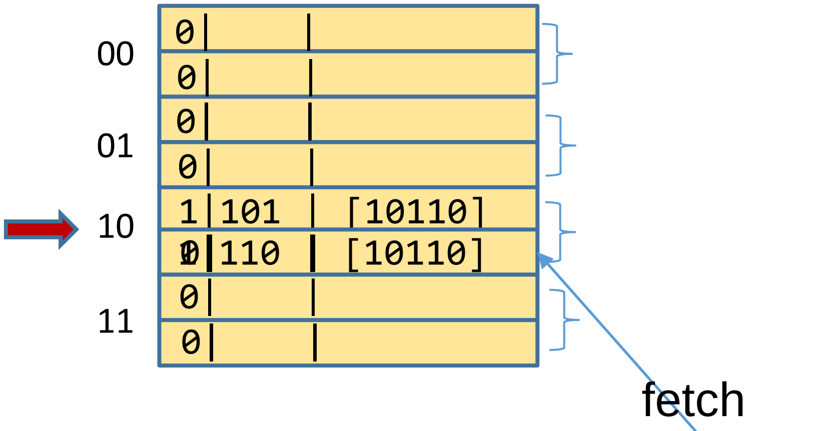


00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Two-Way Set Associative Mapping Example

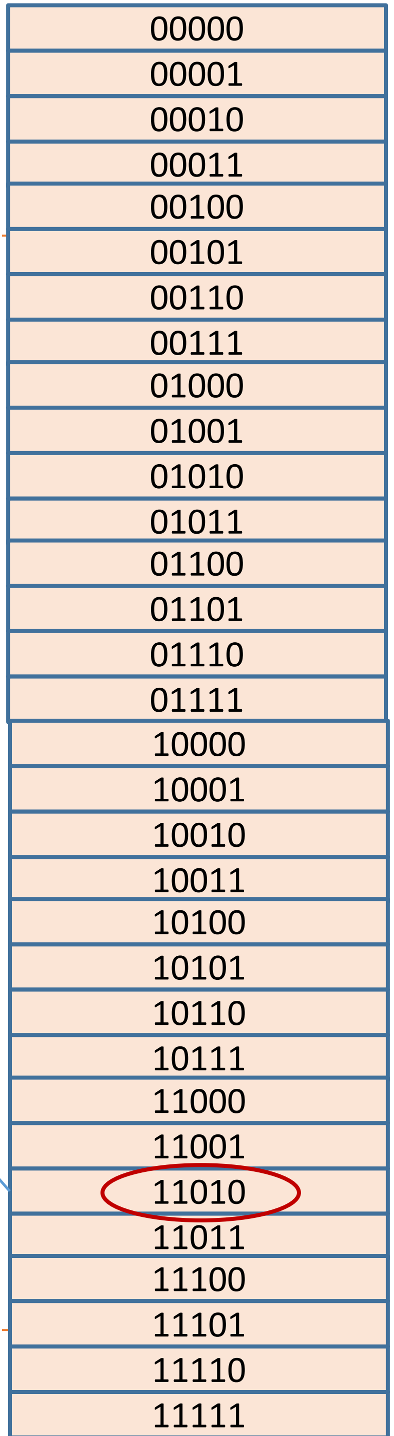
Address block accessed:

- 101 10
- 110 10



00	0		
	0		
01	0		
	0		
10	1	101	[10110]
	1	110	[10110]
11	0		
	0		

fetch

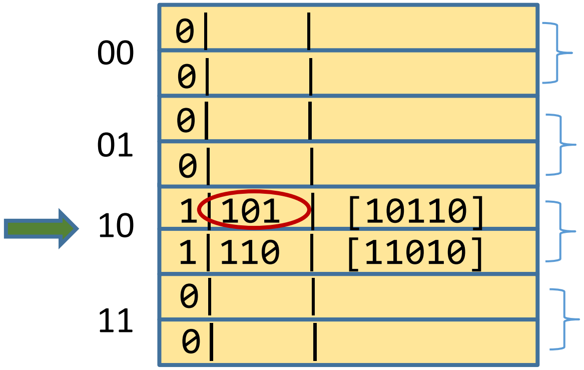


00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Two-Way Set Associative Mapping Example

Address block accessed:

- 101 10
- 110 10
- 101 10

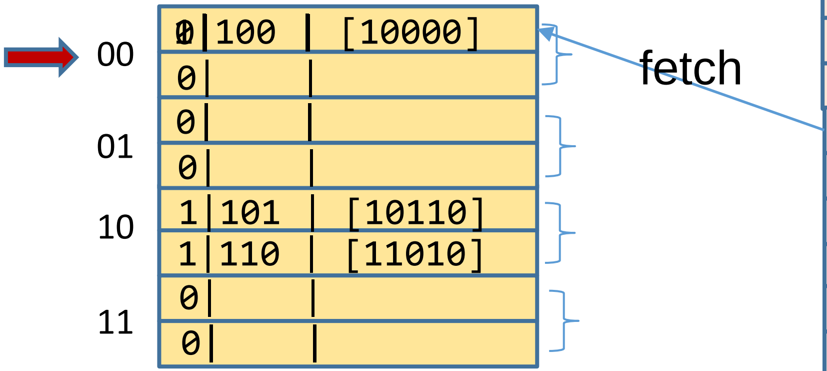


00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Two-Way Set Associative Mapping Example

Address block accessed:

- 101 10
- 110 10
- 101 10
- 100 00



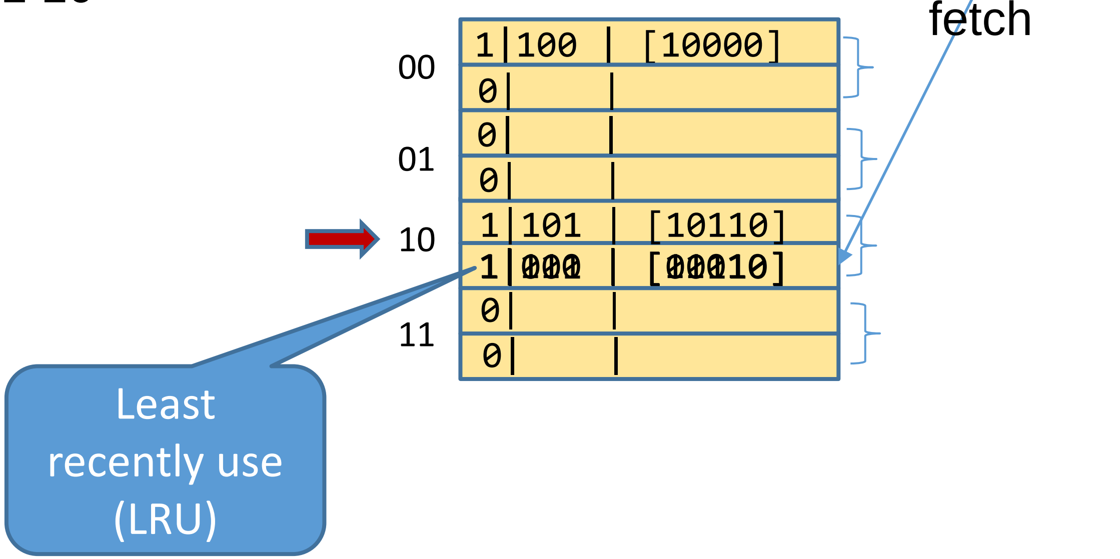
1|100 | [10000]
0| |
0| |
0| |
1|101 | [10110]
1|110 | [11010]
0| |
0| |

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Two-Way Set Associative Mapping Example

Address block accessed:

- 101 10
- 110 10
- 101 10
- 100 00
- 001 10



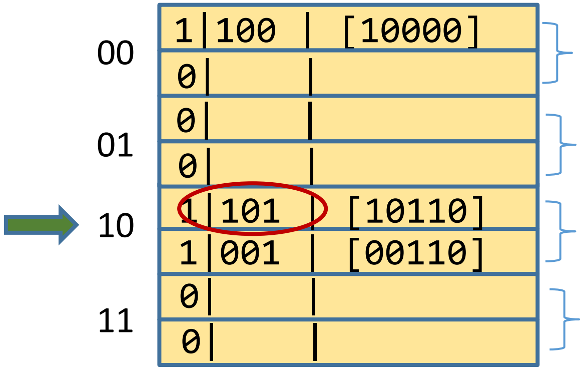
00	1	100	[10000]
	0		
01	0		
	0		
10	1	101	[10110]
	1	000	[00010]
11	0		
	0		

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

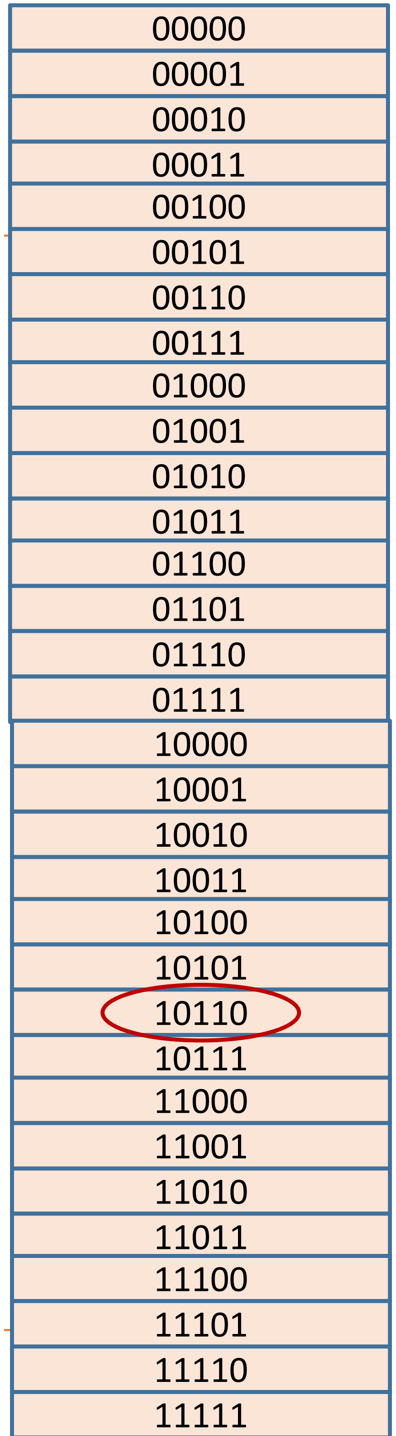
Two-Way Set Associative Mapping Example

Address block accessed:

- 101 10
- 110 10
- 101 10
- 100 00
- 001 10
- 101 10



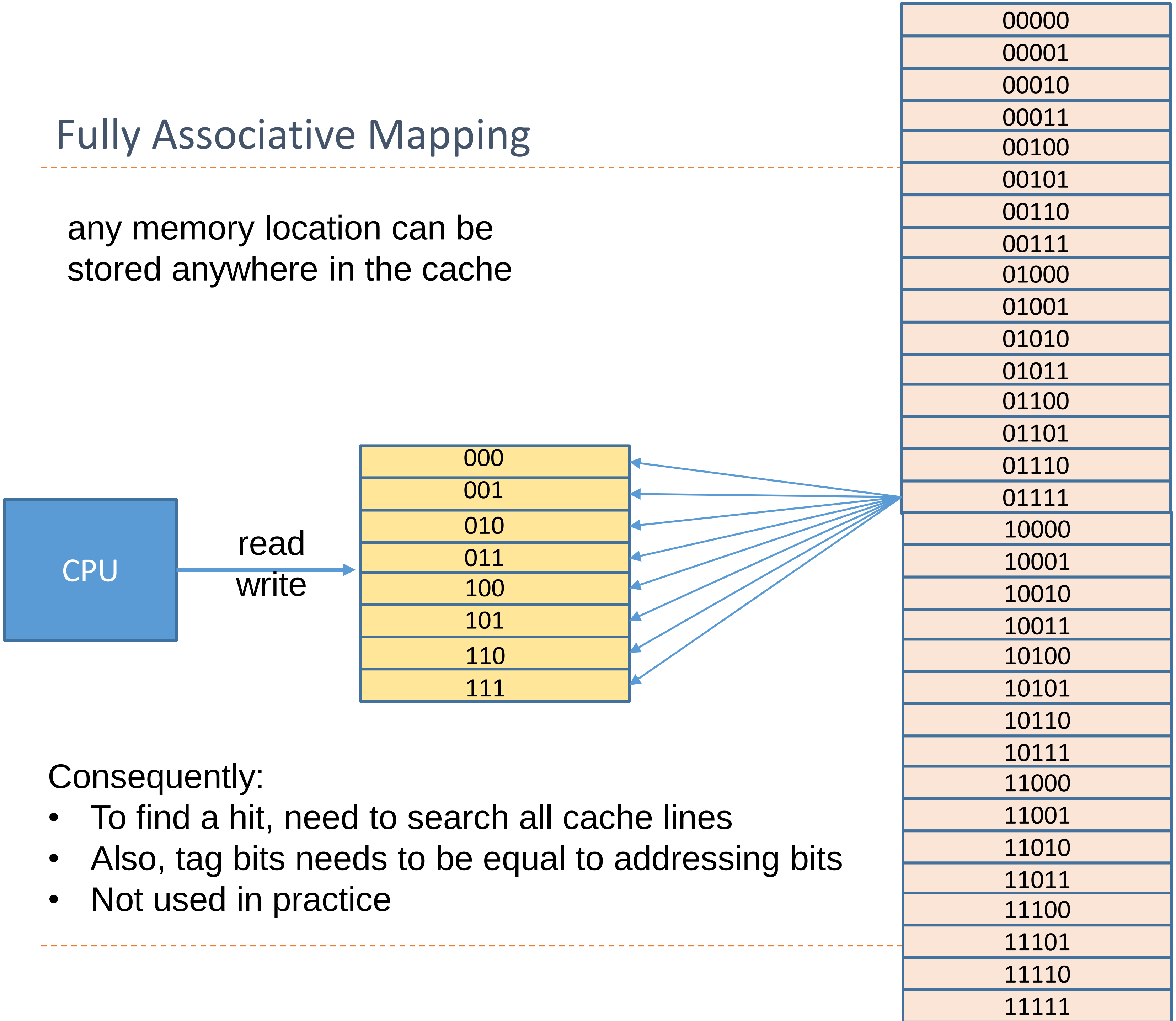
00	1|100 | [10000]
	0| |
01	0| |
	0| |
10	1|101 | [10110]
	1|001 | [00110]
11	0| |
	0| |



00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Fully Associative Mapping

any memory location can be stored anywhere in the cache



Consequently:

- To find a hit, need to search all cache lines
- Also, tag bits needs to be equal to addressing bits
- Not used in practice

Associativity Example

- Three 4 one-word block caches
  - Direct mapped, 2-way set associative, fully associative
- given the following Block access sequence: 0, 8, 0, 6, 8
- Q: Find the number of misses for each organization
- Direct mapped
  - Cache index (0 modulo 4) = 0, (6 modulo 4) = 2, (8 module 4 = 0)

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Note: Blank entry: the block is invalid
Colored text: a new entry added
Plain text: an old entry

Associativity Example

- 2-way set associative
  - Cache set: (0 modulo 2) = 0, (6 modulo 2) = 0, (8 modulo 2) = 0

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative
  - Any memory block can be stored in any cache block

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity

- Increased associativity decreases miss rate
  - If 8 blocks in the cache, two-way associative cache would have the same number of misses as the fully associative cache.
- How much of a reduction in the miss rate is achieved by associativity?
  - Table shows the improvement for a 64KiB D-cache with a 16-word block and associativity ranging from direct-mapped to eight way.

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

Summary of Cache Mapping

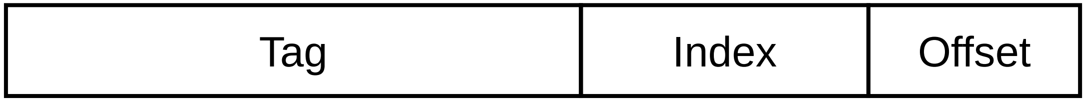
- Direct Mapped
  - A memory block can be placed at a single corresponding location in the cache
  - Fast indexing mechanism
  - Can be viewed as 1-way set associative

- Set-Associative
  - A memory value can be placed in a cache line in a particular set
  - Slightly more complex search mechanism (search within the set)

- Fully-Associative
  - A memory block can be placed in any cache line
  - Extensive hardware resources required to search
    - A comparator associated with each cache entry, which increase the hardware cost
  - Can be viewed as single-set set associative mapping
    - Number of ways equals number of cache lines

Locating a Block in the Cache

Memory Address:

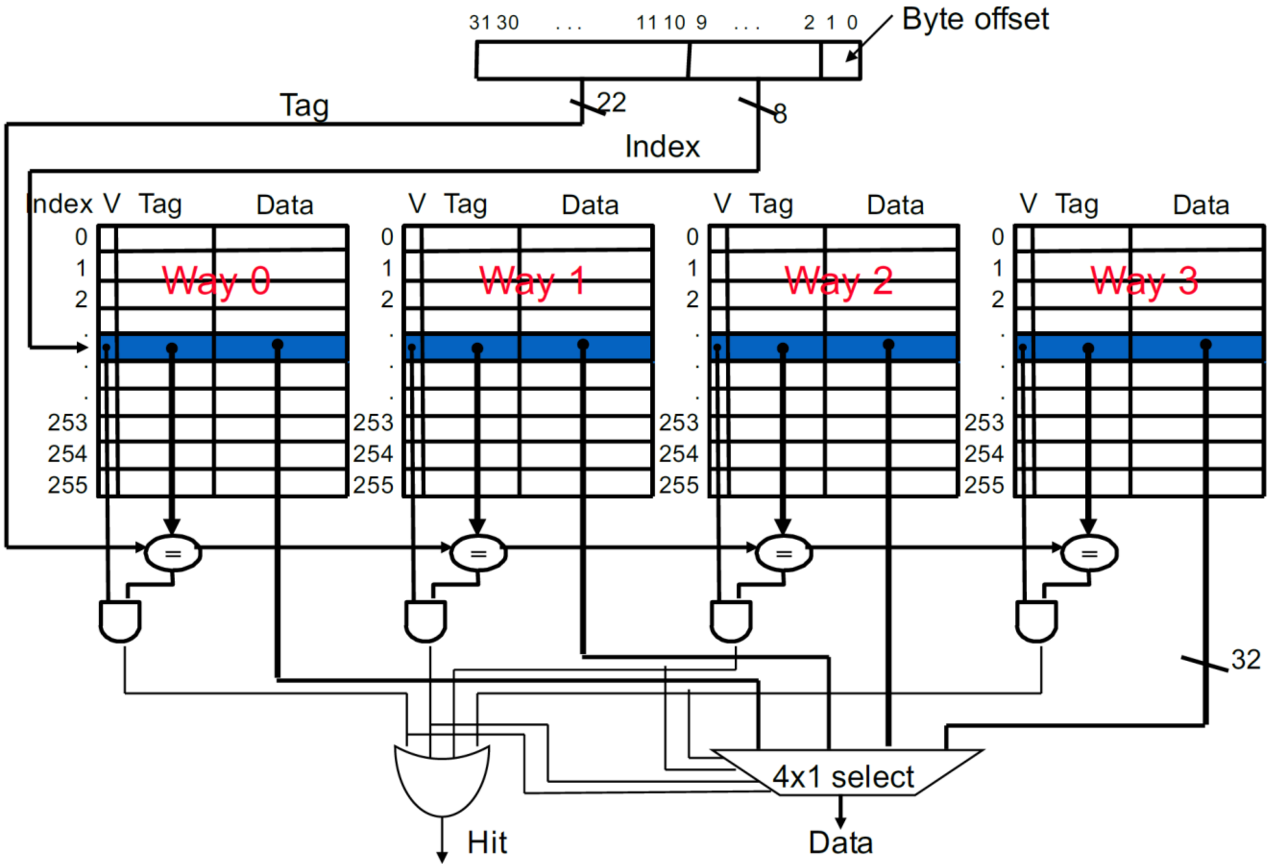


Tag	Index	Offset

- Offset bits
  - Used to locate a particular byte within a cache line
  - Base 2 logarithm of cache line (cache block) size, i.e., $\log_2 \mathit{Size\ of\ cache\ line}$
  - For example, if cache line is 16byte, offset has 4 bits
- Index bits
  - Used to locate a particular set of cache
  - Base-2 logarithm of set numbers, i.e., $\log_2 \mathit{Number\ of sets\ in\ cache}$
  - For example, if there are 8 sets, index has 3 bits
- Tag bits
  - Used for tagging a cache line
  - tag_bits = 32 - index_bits – offset_bits

The implementation of a four-way Set-associative cache

- Four comparators and a 4-to-1 multiplexor
- four tags in the set are compared in parallel



How much associativity depends on the cost of miss v.s. the cost of implementing associativity.

Example: Size of Tags v.s Set Associativity

- Given:

  a 4096 blocks cache, a block size is 16 bytes, and a 32-bit address

Question:

Find the total number of sets and the total number of tag bits for cache that are direct-mapped, two-way and four-way set associative, and fully associative.

Solution:

Offset has $\log_2 16 = 4$ bits

So 32-4 = 28 bits to be used for index and tag.

Direct-mapped: index has $\log_2 4096 = 12$ bits, so tag has 28 – 12 = 16 bits / block

Two-way set associative: there are 4096 / 2 = 2048 sets. Index has $\log_2 2048 = 11$ bits, so tag has 28 – 11 = 17 bits/block

Four-way set associative: there are 4096 / 4 = 1024 sets, index has $\log_2 1024 = 10$ bits, so tag has 28 – 10 = 18 bits/ block

Fully associative: there are one set, so the tag is 28 bits/block.

Increasing associativity requires more comparators and more tag bits per cache block

Cache Example

- Assume separate instruction and data caches
- So we consider only the data
- Cache has space for 8 blocks
- A block contains one word (byte)
- A[10][4] is an array of words located at 7A00-7A27 in row-major order

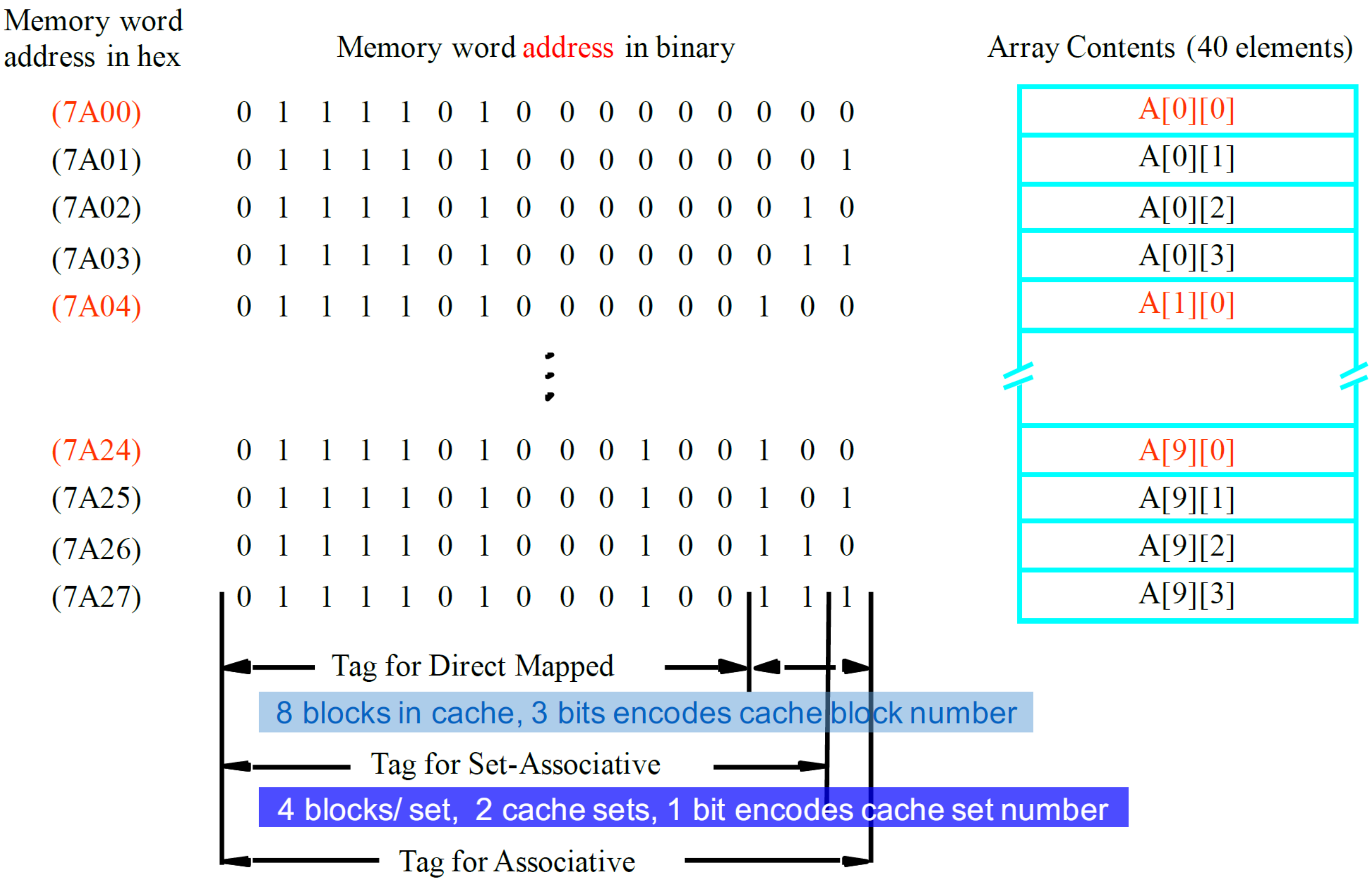
```
short  A[10][4];
int    sum = 0;
int    j, i;
double mean;

// forward loop
for (j = 0; j <= 9; j++)
    sum += A[j][0];

mean = sum / 10.0;

// backward loop
for (i = 9; i >= 0; i--)
    A[i][0] = A[i][0]/mean;
```

Cache Example



Memory word address in hex	Memory word address in binary	Array Contents (40 elements)
(7A00)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0	A[0][0]
(7A01)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1	A[0][1]
(7A02)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 0	A[0][2]
(7A03)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 1	A[0][3]
(7A04)	0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0	A[1][0]
	⋮	
(7A24)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 0	A[9][0]
(7A25)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1	A[9][1]
(7A26)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0	A[9][2]
(7A27)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 1	A[9][3]

Tag for Direct Mapped

8 blocks in cache, 3 bits encodes cache block number

Tag for Set-Associative

4 blocks/ set, 2 cache sets, 1 bit encodes cache set number

Tag for Associative

To simplify discussion: 16-bit word (byte) address; i.e. 1 word = 1 byte

Direct Mapping

- Least significant 3-bits of address determine location
- No replacement algorithm is needed in Direct Mapping
- When $i == 9$ and $i == 8$, get a cache hit (2 hits in total)
- Only 2 out of the 8 cache positions used
- Very inefficient cache utilization

		Content of data cache after loop pass: (time line)																			
		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
Cache Block number	0	A[0][0]	A[0][0]	A[2][0]	A[2][0]	A[4][0]	A[4][0]	A[6][0]	A[6][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[6][0]	A[6][0]	A[4][0]	A[4][0]	A[2][0]	A[2][0]	A[0][0]
	1																				
	2																				
	3																				
	4		A[1][0]	A[1][0]	A[3][0]	A[3][0]	A[5][0]	A[5][0]	A[7][0]	A[7][0]	A[9][0]	A[9][0]	A[9][0]	A[7][0]	A[7][0]	A[5][0]	A[5][0]	A[3][0]	A[3][0]	A[1][0]	A[1][0]
	5																				
	6																				
	7																				

Associative Mapping

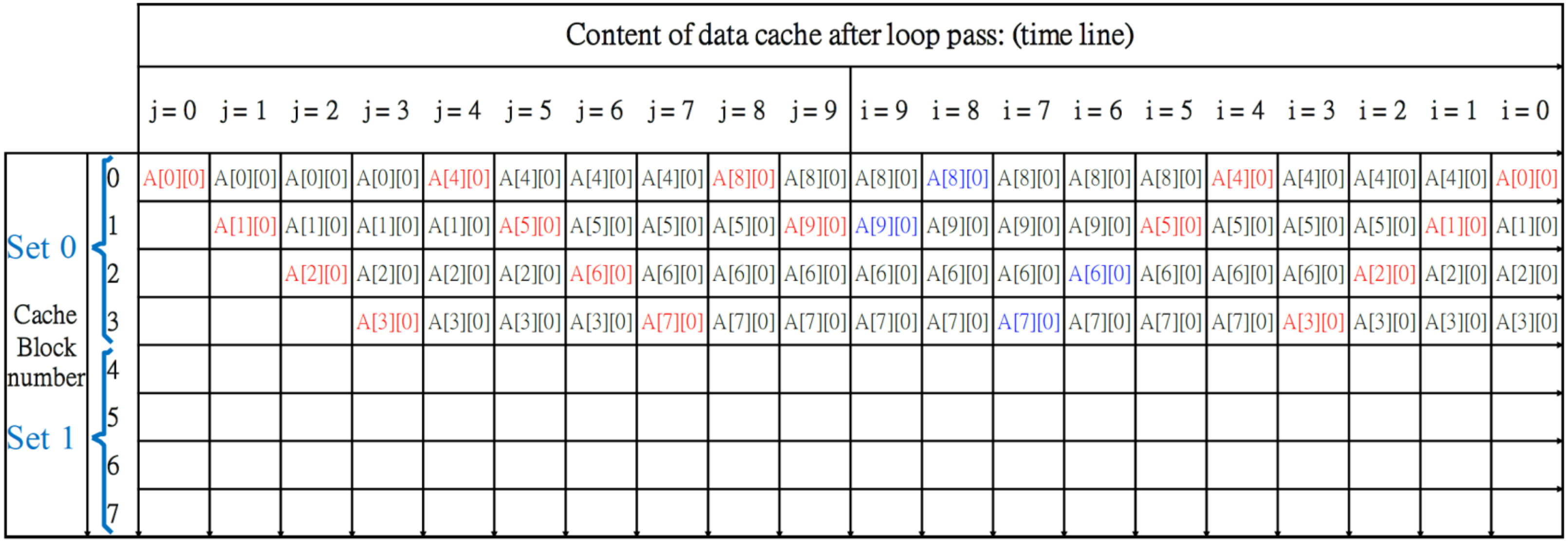
- LRU replacement policy: get cache hits for $i = 9, 8,.., 2$
- If i loop was a forward one, we would get no hits!

		Content of data cache after loop pass: (time line)																			
		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
Cache Block number	0	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[0][0]
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[1][0]	A[1][0]
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]
	4					A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]
	5						A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]
	6							A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]
	7								A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]

Tags not shown but are needed; LRU Counters not shown but are needed

Set associative Mapping

- Since all accessed blocks have even addresses (7A00, 7A04, 7A08, ...), only half of the cache is used, i.e. they all map to set 0
- LRU replacement policy: get hits for i = 9, 8, 7 and 6
- Random replacement would have better average performance
- If i loop was a forward one, we would get no hits!



	Cache Block number	Content of data cache after loop pass: (time line)																			
		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
Set 0	0	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[0][0]
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[1][0]	A[1][0]
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[2][0]	A[2][0]	A[2][0]
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]
Set 1	4																				
	5																				
	6																				
	7																				

Tags not shown but are needed; LRU Counters not shown but are needed

Comments on the Example

- In this example, Associative is best, then Set-Associative, lastly Direct Mapping.
- What are the advantages and disadvantages of each scheme?
- In practice,
  - Low hit rates like in the example is very rare.
  - Usually Set-Associative with LRU replacement scheme is used.
- Larger blocks and more blocks greatly improve cache hit rate, i.e. more cache memory

Handling the Cache Read

- On cache read hit
  - CPU proceeds normally
- On cache read miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - install it in the cache and send the requested word to the processor
  - then let the pipeline resume
    - Instruction cache miss
      - □ Restart instruction fetch
    - Data cache miss
      - □ Complete data access

Handling Writes

- On data-write hit
  - could just update the block in cache
  - But then cache and memory would be inconsistent
  - Simplest way-Write through to avoid inconsistent
  - Write through: Write the data into both the memory and the cache.
- On a data write miss
  - First fetch the words of the block from memory into the cache.
  - Overwrite the word in the cache block.
  - All write the word to main memory avoiding inconsistent.
- Very simply, but not good performance.
  - Every write cause the data to be written to main memory.
  - But this writes take a longer time, at least 100 clock cycles
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11, reducing performance by more than a factor of 10.

Handling Writes

- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues execution after writing the data into the cache and into the write buffer.
    - Only stalls on write if write buffer is already full until there is empty position in the write buffer
  - When a write to main memory completes, the entry in the write buffer is freed.

- Alternative Solution: Write -back
  - just update the data of the block in cache
  - Overwrite the data to the lower level of the hierarchy when the block is replaced.
  - Improve the performance, however, more complex to implement than write-through

More detail on Chapter 5.8

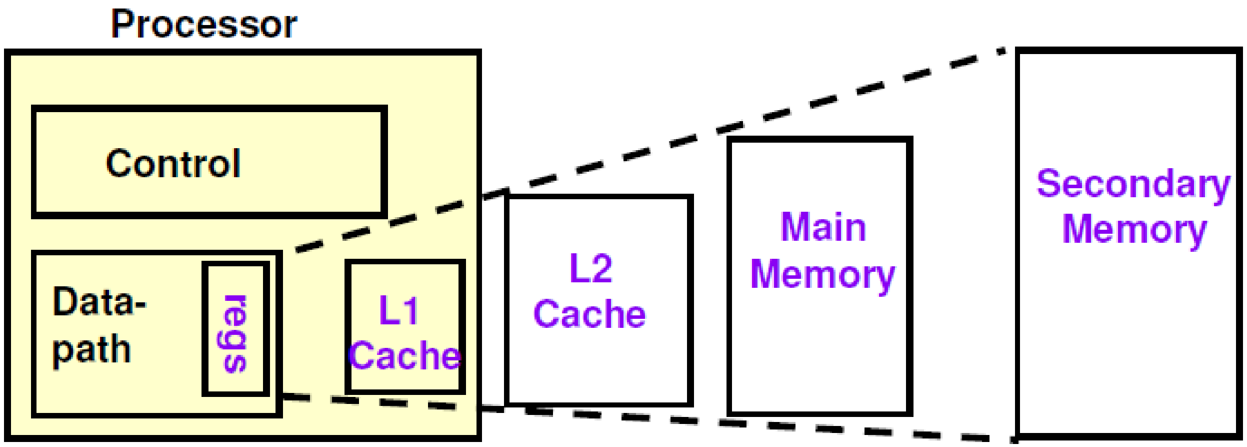
Replacement Policy

When a miss occurs,

- Direct mapped:
  - Position of each block fixed
  - the choice is obvious and thus no “replacement algorithm” is needed
- Associative & Set associative: choose which block to replace
  - Strategy 1. Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time.
    - e.g. for a 4-block/set cache, use a $\log_2 4 = 2$ bits counter for each block
    - Reset the counter to 0 whenever the block is accessed
    - counters of other blocks in the same set should be incremented
    - On cache miss, replace/ uncache a block with counter reaching 3
  - Strategy 2. Random Replacement
    - Choose random block
    - Easier to implement at high speed

Reducing the Miss Penalty Using Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Second-Level cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services Second-Level cache misses
- Some high-end systems include L-3 cache



Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- How much faster will the processor be if we add a secondary cache that has a 5-ns access time for either a hit or a miss and is large enough to reduce the miss rate to main memory to 0.5%
- With just primary cache
  - Clock cycle time = 1/4GHz = 0.25ns
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache, a miss in the primary cache can be satisfied by the L-2 cache or by the main memory.
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

A Two-Level Cache Considerations

- Primary cache
  - Focus on minimal hit time to yield a shorter clock cycle
- L-2 cache
  - Focus on low miss rate to reduce the penalty of long memory access times.
  - Hit time has less overall impact
- In comparison to a single-level cache
  - L-1 cache usually is smaller than a single cache
  - L-1 may use smaller block size to reduce the miss penalty
  - L-1 block size smaller than L-2 block size
  - L-2 cache much larger than in a single-level cache, a larger block size, higher associativity than L-1 cache to reduce miss rate.

A Cache Example: The Intrinsity FastMATH processor

- Split cache: separate I-cache and D-cache
- Each cache has 16KB
- Each cache block has 16 words (64 bytes)
- Direct mapping

- Q: How many blocks for each cache?

$$\frac{cache\ size}{cache\ block\ size} = \frac{16 \times 1024}{16 \times 4} = 256$$

- Q: How many offset bits? Index bits? And tag bits?

18 bits	8 bits	6 bits
tag	index	offset
32 – 6 - 8	$\log_2 256$	$\log_2 64$

A Cache Example: The Intrinsity FastMATH processor

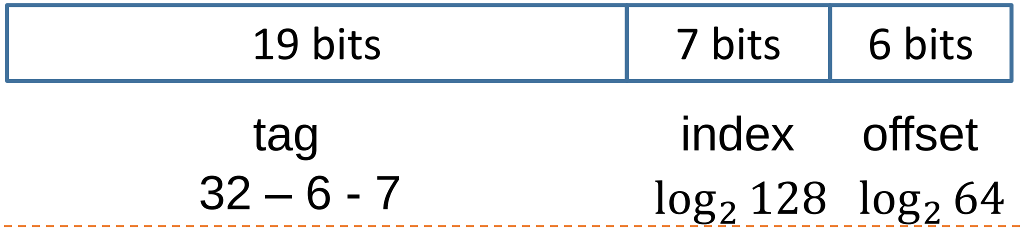
- Suppose in the latest design, designer likes to keep the total cache size, but change mapping to 2-way set associative mapping

- Q: How many cache blocks in each set?
  - 2

- Q: How many sets for each cache?

$$\frac{total\ cache\ size}{cache\ block\ size \times way} = \frac{16\times1024}{16\times4\times2} = 128$$

- Q: How many bits in offset, index and tag field?



A Cache Example: The Intrinsity FastMATH processor

- In the last design, assume CPU wants to access byte at address 0x03031515
- Q: Which set of cache CPU should search for? That is, what is the set index?
- Q: assume it is a cache hit, which byte inside the cache line is what CPU wants? That is, what is the offset inside the cache line?

Converting address 0x03031515 to binary representation

0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1

Index: 84

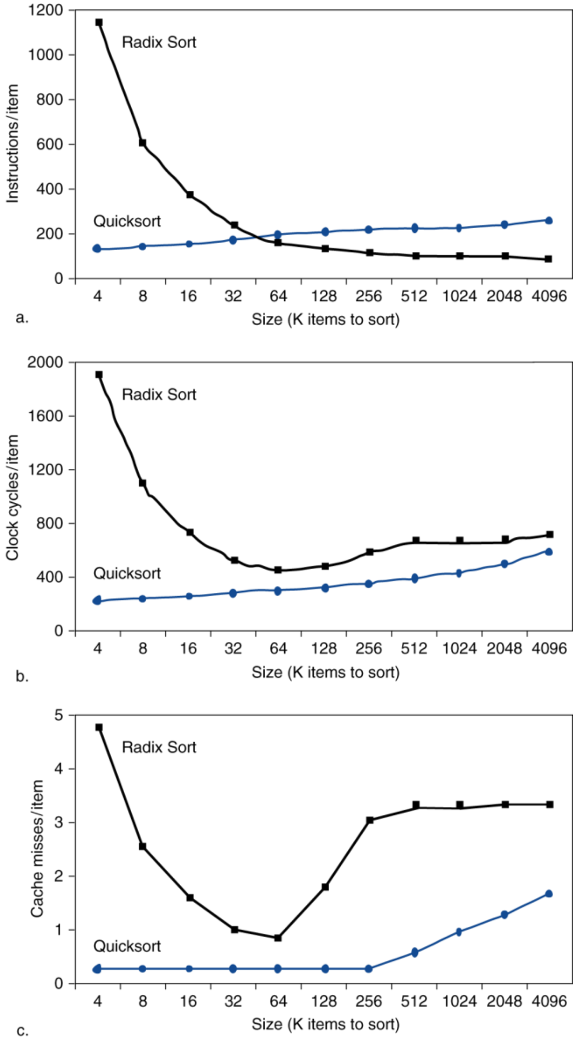
Offset: 21

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation

Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access



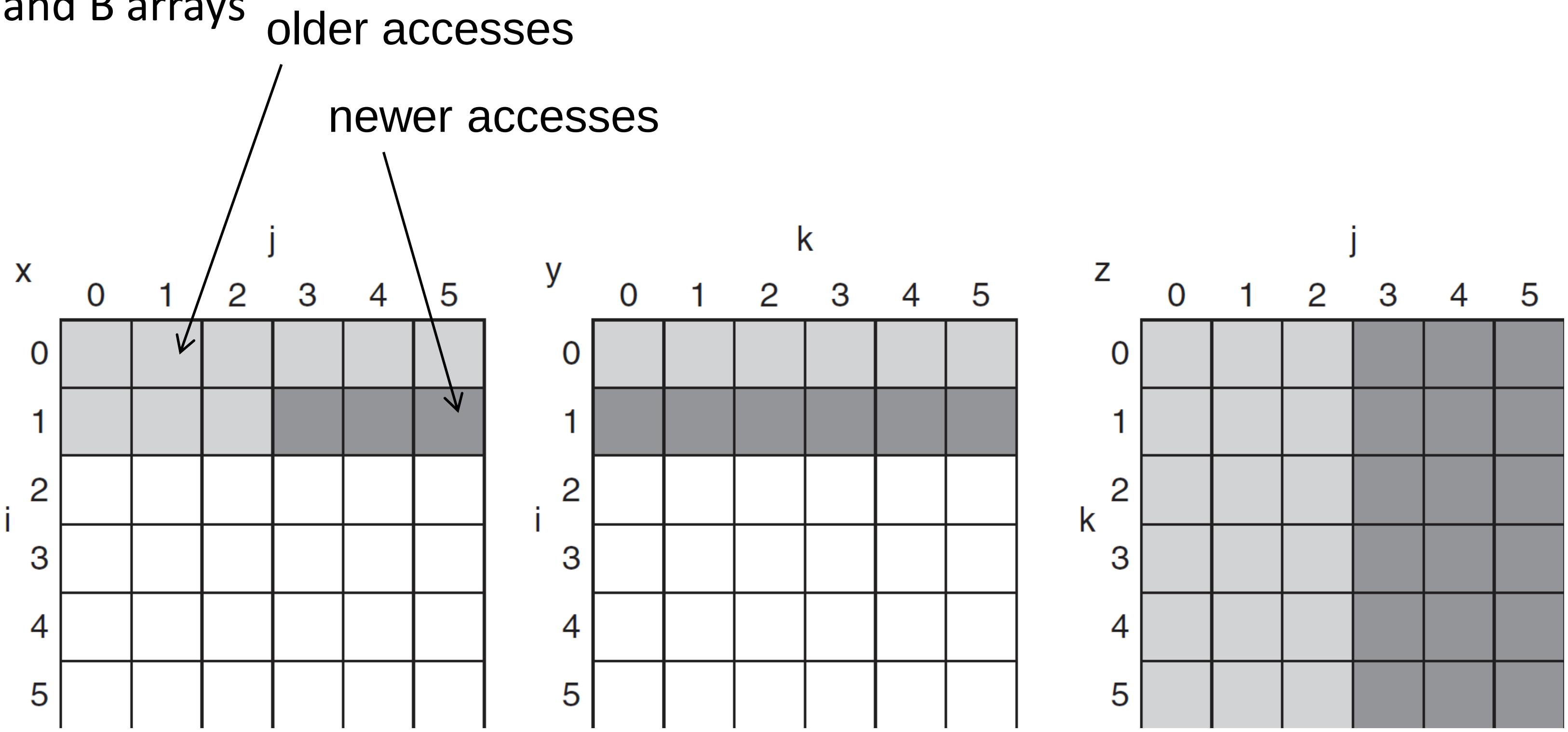
Software Optimization via Blocking

- Goal: maximize accesses to the data loaded into the cache before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```

DGEMM Access Pattern

- C, A, and B arrays

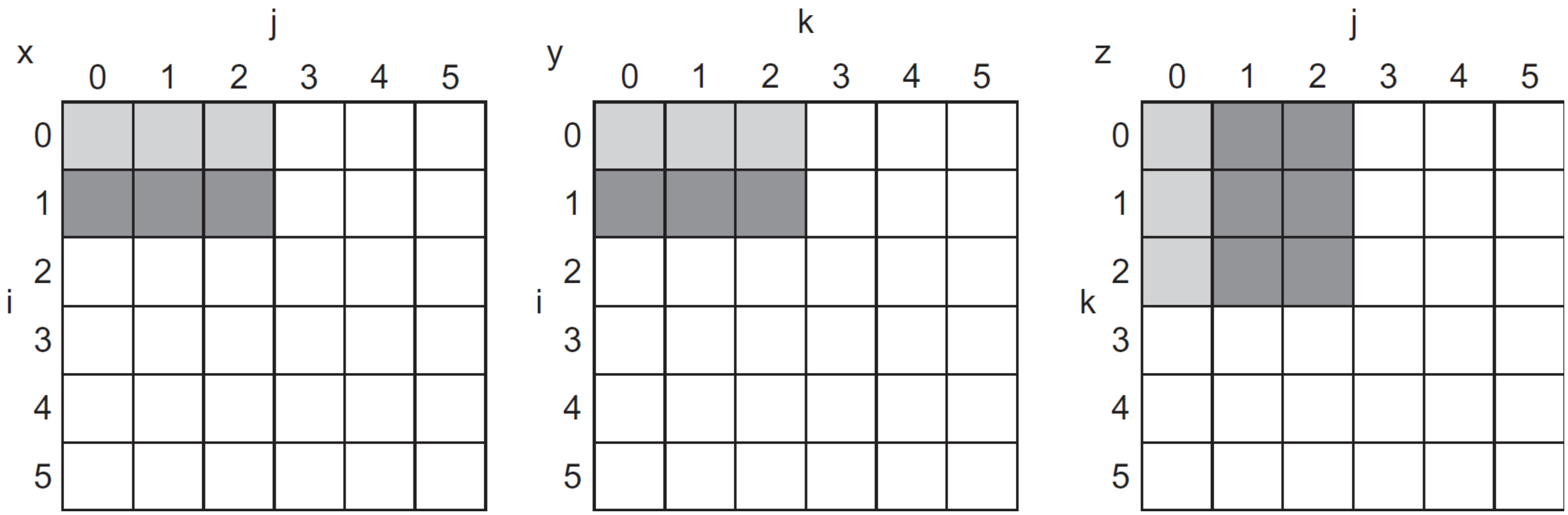


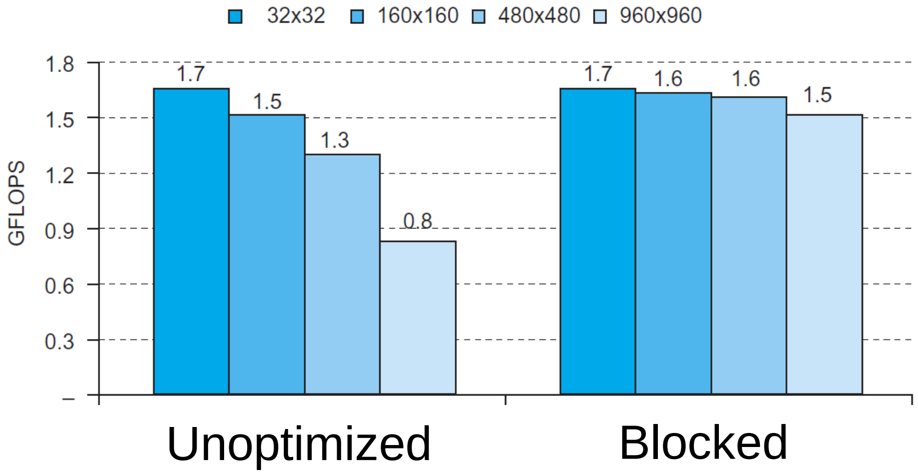
- If the cache can hold one N-by-N matrix and one row of N, then at least the ith row of A and the array B may stay in the cache. less than that and misses may occur for both B and C.
- To ensure the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix.

Cache Blocked DGEMM

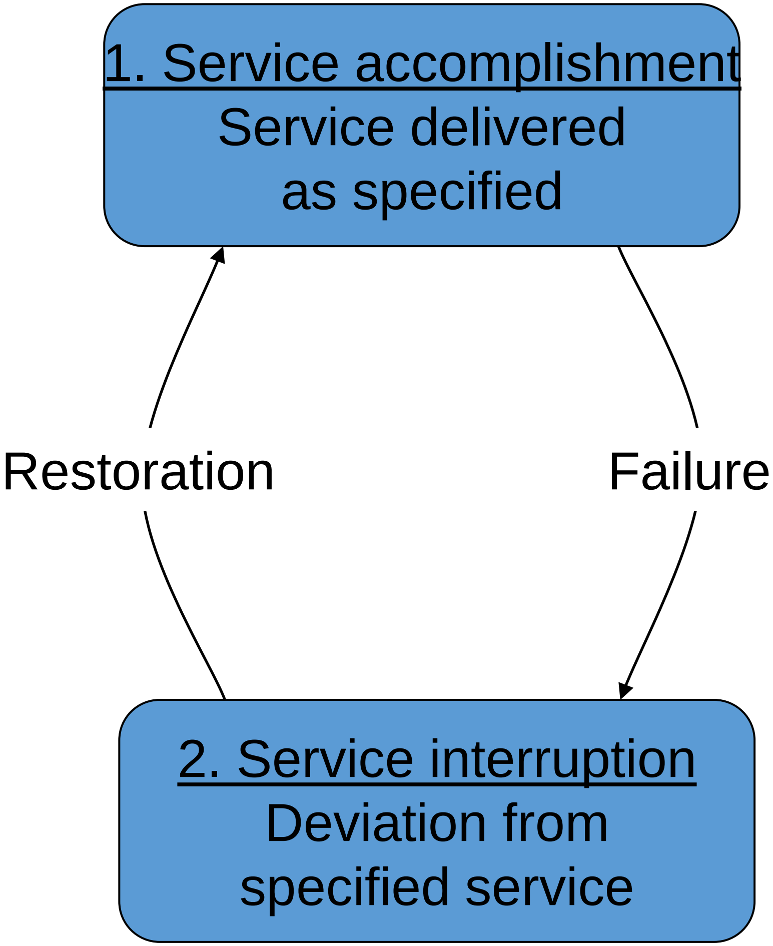
```
#define BLOCKSIZE 32
void do_block (int n, int si, int sj, int sk, double *A, double
*B, double *C)
{
 for (int i = si; i < si+BLOCKSIZE; ++i)
  for (int j = sj; j < sj+BLOCKSIZE; ++j)
  {
   double cij = C[i+j*n];/* cij = C[i][j] */
   for( int k = sk; k < sk+BLOCKSIZE; k++ )
    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
   C[i+j*n] = cij;/* C[i][j] = cij */
  }
 }
 void dgemm (int n, double* A, double* B, double* C)
 {
  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
   for ( int si = 0; si < n; si += BLOCKSIZE )
    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
     do_block(n, si, sj, sk, A, B, C);
 }
```

Blocked DGEMM Access Pattern





Dependability



1. Service accomplishment
Service delivered
as specified

Restoration

Failure

2. Service interruption
Deviation from
specified service

Fault: failure of a component which may or may not lead to system failure

Dependability Measures

- Reliability:mean time to failure (MTTF)
  - a related term: annual failure rate(AFR)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF:
    - fault avoidance: preventing fault occurrence by construction
    - fault tolerance: Using redundancy to allow the service to comply with the service specification despite fault occurring.
    - fault forecasting: predicting the presence and creation of faults, allowing the components to be replace before it fails

The Hamming SEC Code

- Hamming distance
  - The minimum number of bits that are different between any two correct bit patterns
  - E.g. the distance between 011011 and 001111 is two
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code for error detection
    - Odd parity: the number of 1s is odd
    - Even parity: the number of 1s is even
    - The parity bit is stored for each word: 1 for odd, 0 for even.
    - The N+1 bit word should always be even.
    - If the parity of the word and the stored parity bit do not match, an error has occurred.
- Minimum distance = 3 provides single error correction, 2 bit error detection
- We call hamming error correction code(ECC)

Encoding SEC

- To calculate Hamming ECC:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - ALL other bit positions are used for data bits.
  - Each parity bit checks certain data bits:
  - Set parity bits to create even parity for each group.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 (which is 10 in decimal form) indicates bit 10 (d6) was flipped

SEC/DED Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4 SEC and DED
- Decoding:
  - Let H = SEC parity bits
    - H even, p_n even, no error
    - H odd, p_n odd, correctable single bit error
    - H even, p_n odd, error in p_n bit
    - H odd, p_n even, double error occurred
- Note: ECC DRAM uses SEC/DED with 8 bits protecting each 64 bits, which is why many DIMMs are 72 bits wide

Reference Readings

- Patterson, "Computer Organization And Design"
  - Sec. 5.3 ~ 5.5
- Bryant, "Computer Systems"
  - Sec 6.1 ~ 6.4