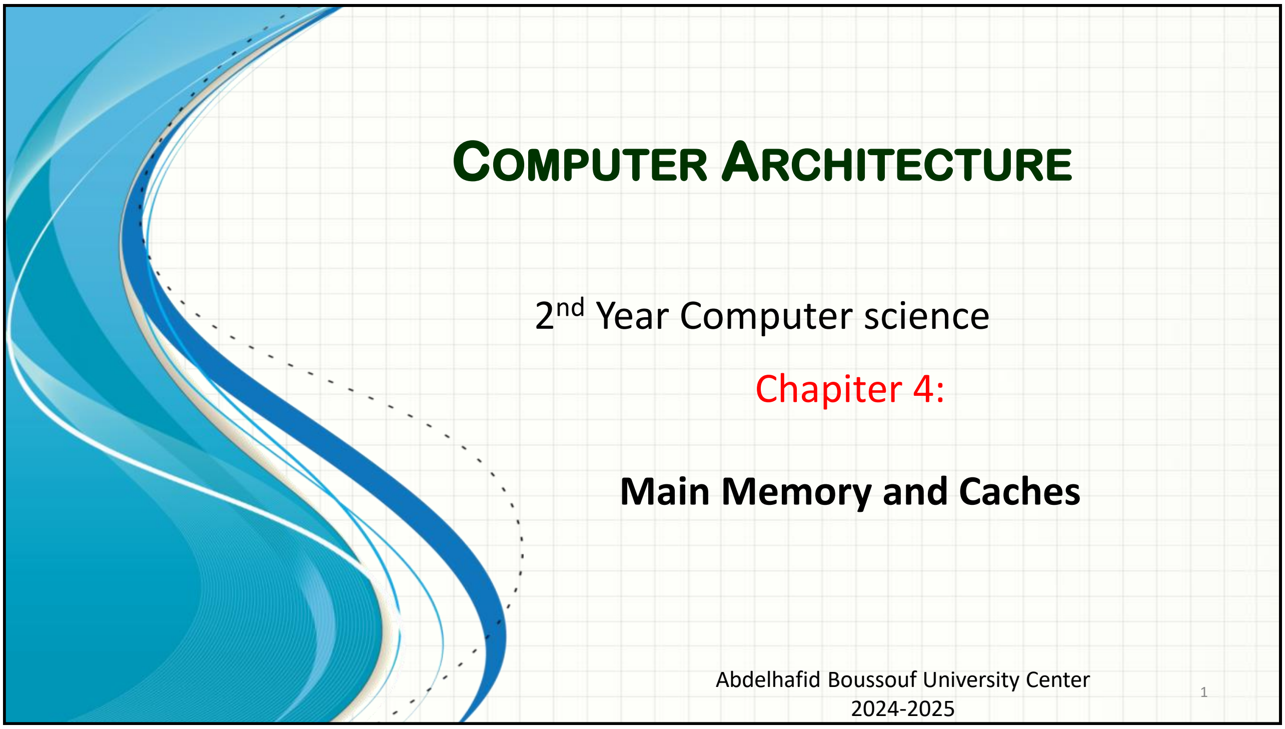


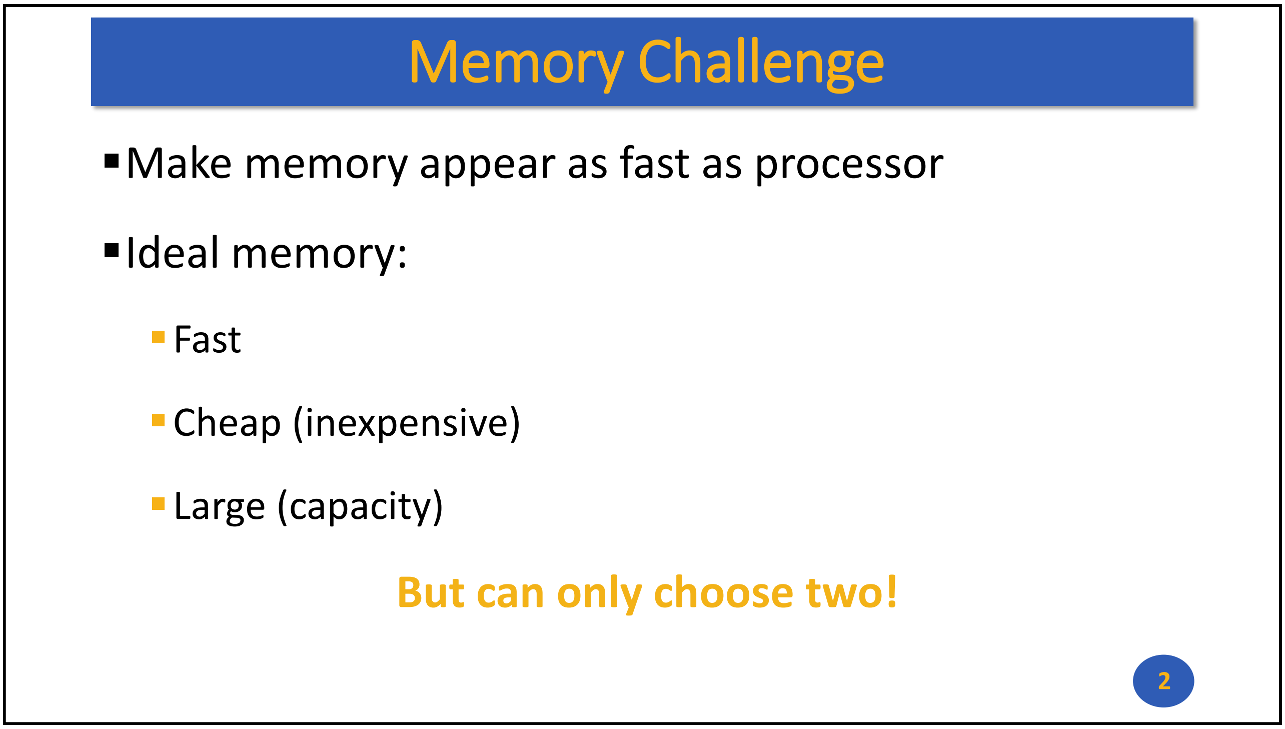
COMPUTER ARCHITECTURE

2nd Year Computer science

Chapiter 4:

Main Memory and Caches

Abdelhafid Boussouf University Center
2024-2025



Memory Challenge

- Make memory appear as fast as processor
- Ideal memory:
  - Fast
  - Cheap (inexpensive)
  - Large (capacity)

But can only choose two!

Review: What were Memory Elements ?

- **Memories are large blocks**
  - A significant portion of a modern circuit is memory.
- **Memories are practical tools for system design**
  - Programmability, reconfigurability all require memory
- **Allows you to store data and work on stored data**
  - Not all algorithms are designed to process data as it comes, some require data to be stored.
  - Data type determines required storage

How Can We Store Data

- **Flip-Flops (or Latches)**
  - Very fast, parallel access
  - Expensive (one bit costs 20+ transistors)
- **Static RAM**
  - Relatively fast, only one data word at a time
  - Less expensive (one bit costs 6 transistors)
- **Dynamic RAM**
  - Slower, reading destroys content (refresh), one data word at a time, needs special process
  - Cheaper (one bit is only a transistor)
- **Other storage technology (hard disk, flash)**
  - Much slower, access takes a long time, non-volatile
  - Per bit cost is lower (no transistors directly involved)

Locality

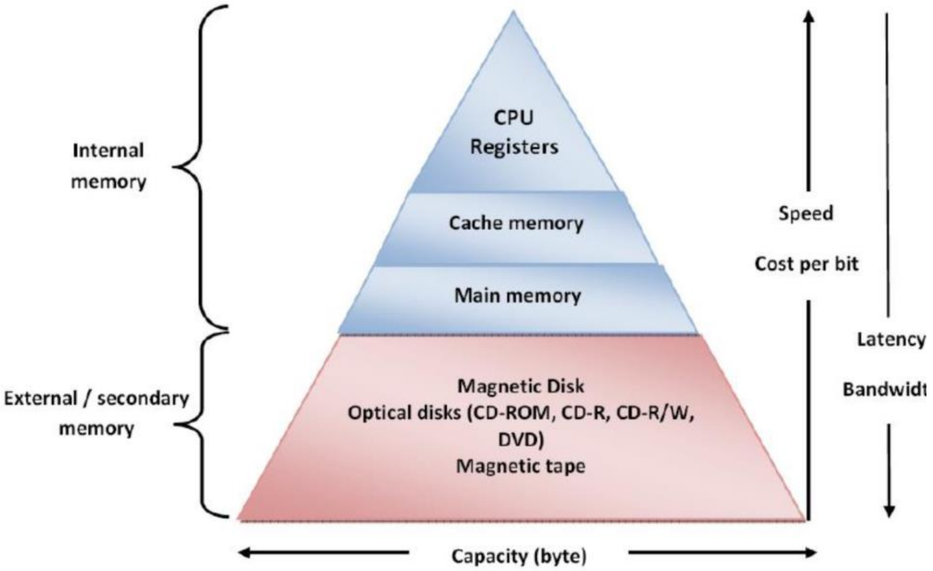
Exploit locality to make memory accesses fast

- **Temporal Locality:**
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
- **Spatial Locality:**
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

Taking Advantage of Locality

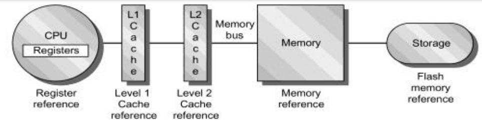
- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU

Memory Hierarchy



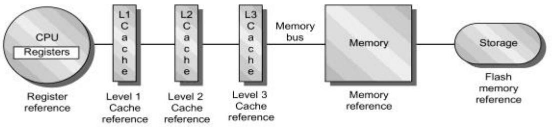
Memory Hierarchy

- Personal mobile device



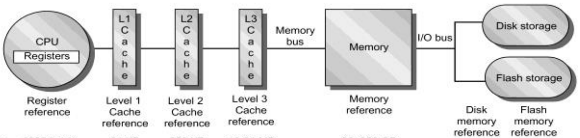
Size:	1000 bytes	64 KB	256 KB	1–2 GB	4–64 GB
Speed:	300 ps	1 ns	5-10 ns	50–100 ns	25–50 us

- Laptop or desktop



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

- Server



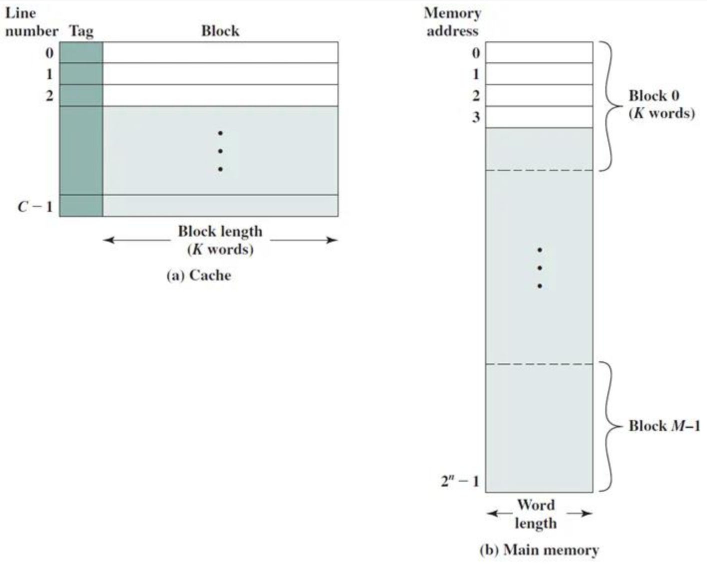
Size:	4000 bytes	64 KB	256 KB	16-64 MB	32–256 GB	16–64 TB	1-16 TB
Speed:	200 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	5–10 ms	100-200 us

rver

Cache Terminology

- **Capacity (*C*):**
  - the number of data bytes a cache stores
- **Block size (*b*):**
  - bytes of data brought into cache at once
- **Number of blocks (*B = C/b*):**
  - number of blocks in cache: *B* = *C*/*b*
- **Degree of associativity (*N*):**
  - number of blocks in a set
- **Number of sets (*S = B/N*):**
  - each memory address maps to exactly one cache set

How It Works?

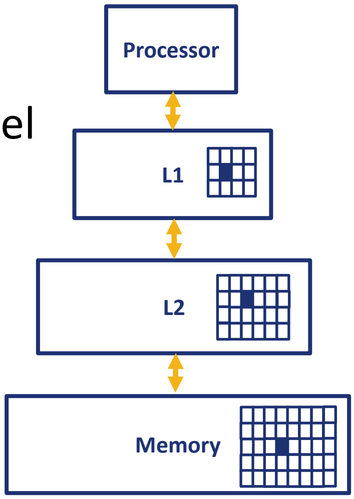


(a) Cache

(b) Main memory

How It Works?

- **Block** (aka **line**): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - **Hit**: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - **Miss**: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level



Hits and Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Miss Types

- **Compulsory**: first time data accessed
- **Capacity**: cache too small to hold all data of interest
- **Conflict**: data of interest maps to same location in cache

Mapping function

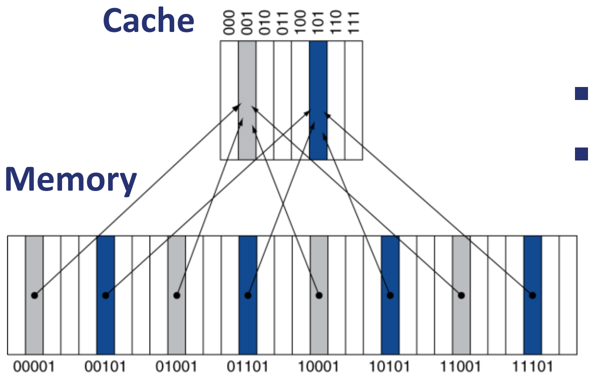
- There are fewer cache blocks than main memory blocks, an algorithm is needed for mapping main memory blocks into cache blocks.
- Need to determine which main memory block currently occupies a cache block.
- The choice of the mapping function dictates how the cache is organized.
- •Three techniques for mapping function:
  - ***Direct mapped***
  - ***N-way set associative***
  - ***Fully associative***

Tags and Valid Bits

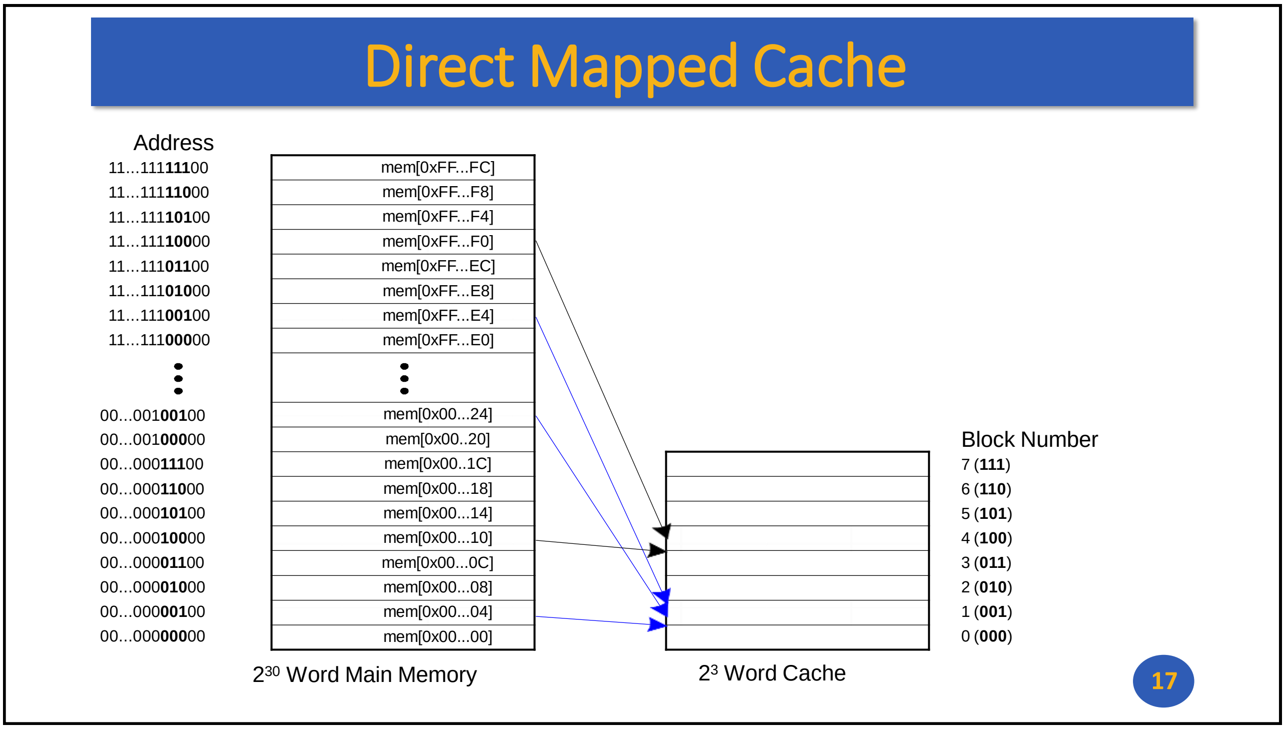
- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block number) modulo (#Blocks in cache)



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



Direct Mapped Cache

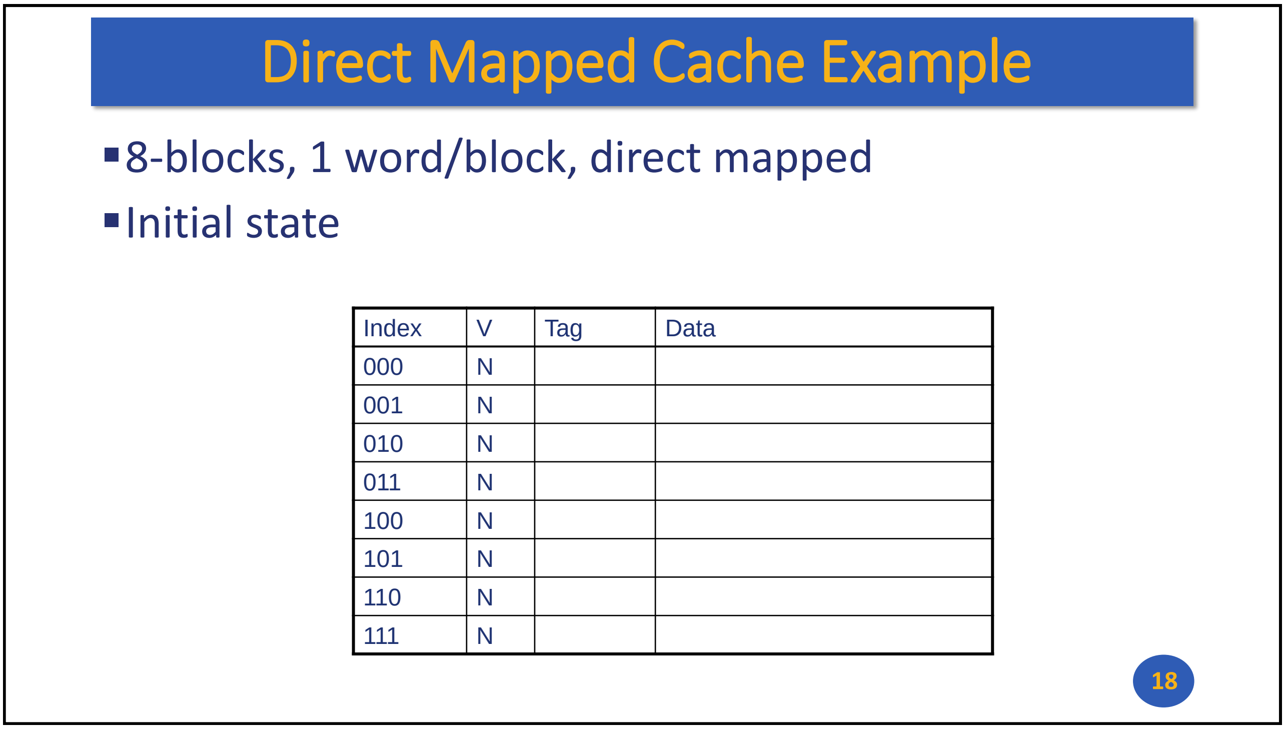
Address	2^30 Word Main Memory
11...111**111**00	mem[0xFF...FC]
11...111**110**00	mem[0xFF...F8]
11...111**101**00	mem[0xFF...F4]
11...111**100**00	mem[0xFF...F0]
11...111**011**00	mem[0xFF...EC]
11...111**010**00	mem[0xFF...E8]
11...111**001**00	mem[0xFF...E4]
11...111**000**00	mem[0xFF...E0]
⋮	⋮
00...001**001**00	mem[0x00...24]
00...001**000**00	mem[0x00..20]
00...000**111**00	mem[0x00..1C]
00...000**110**00	mem[0x00...18]
00...000**101**00	mem[0x00...14]
00...000**100**00	mem[0x00...10]
00...000**011**00	mem[0x00...0C]
00...000**010**00	mem[0x00...08]
00...000**001**00	mem[0x00...04]
00...000**000**00	mem[0x00...00]

2^{30} Word Main Memory

Block Number

7 (**111**)
6 (**110**)
5 (**101**)
4 (**100**)
3 (**011**)
2 (**010**)
1 (**001**)
0 (**000**)

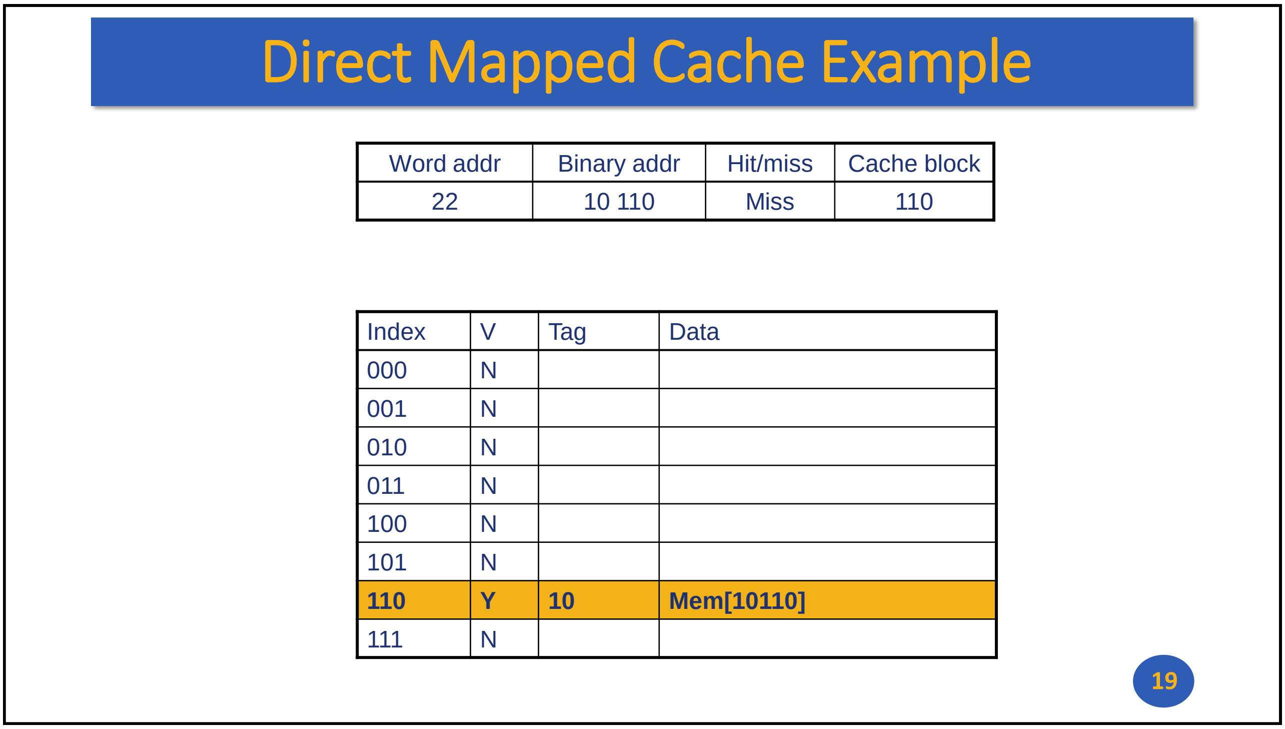
2^3 Word Cache



Direct Mapped Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

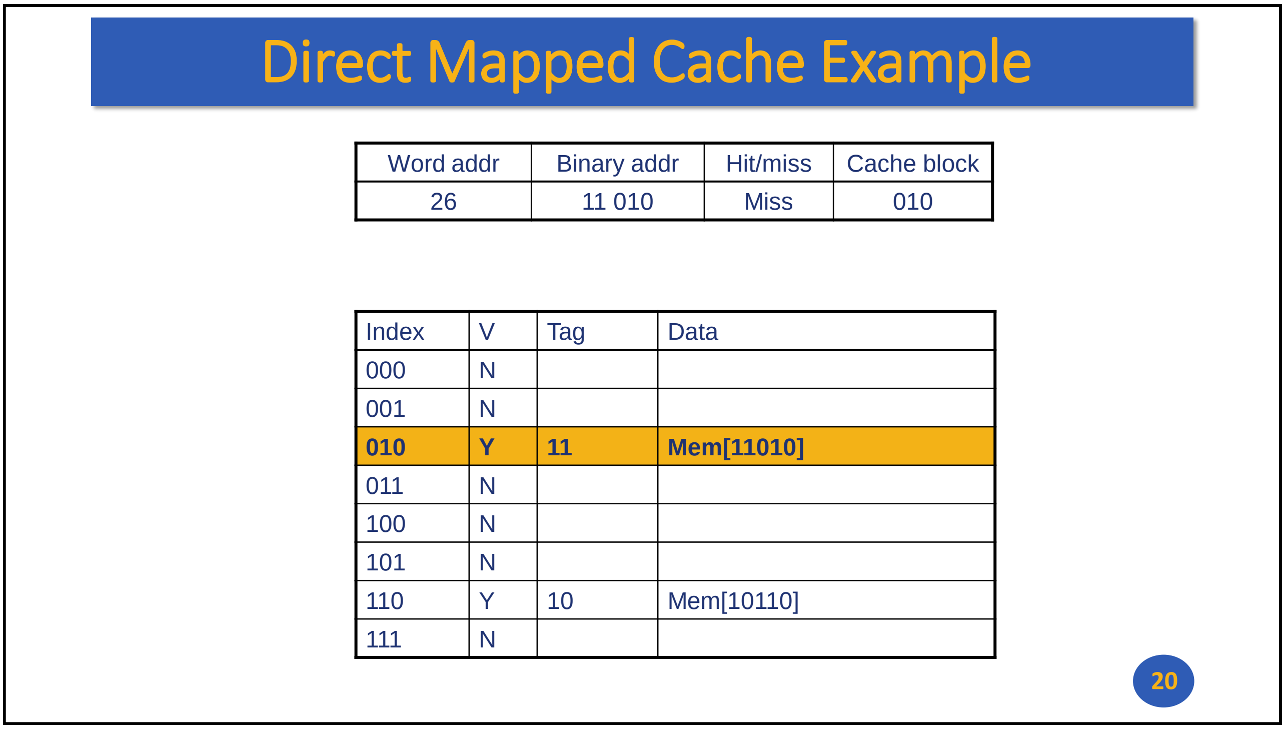
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		



Direct Mapped Cache Example

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

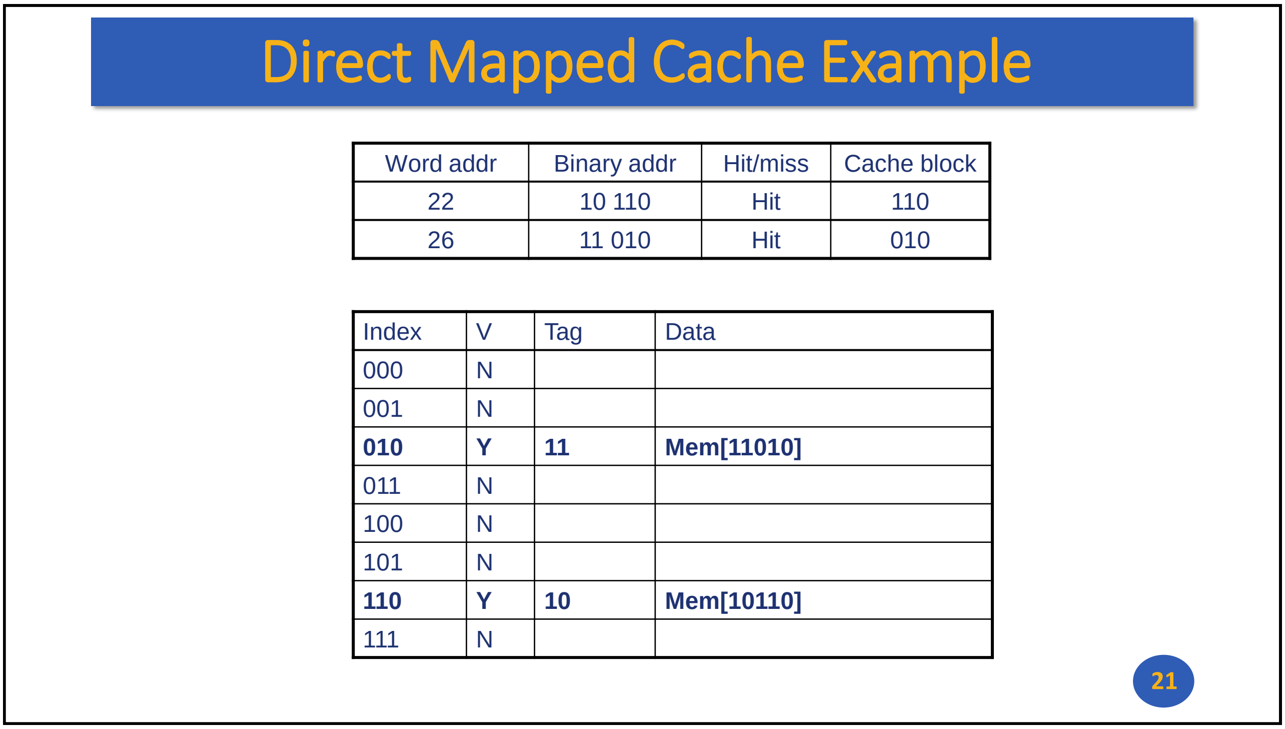
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		



Direct Mapped Cache Example

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

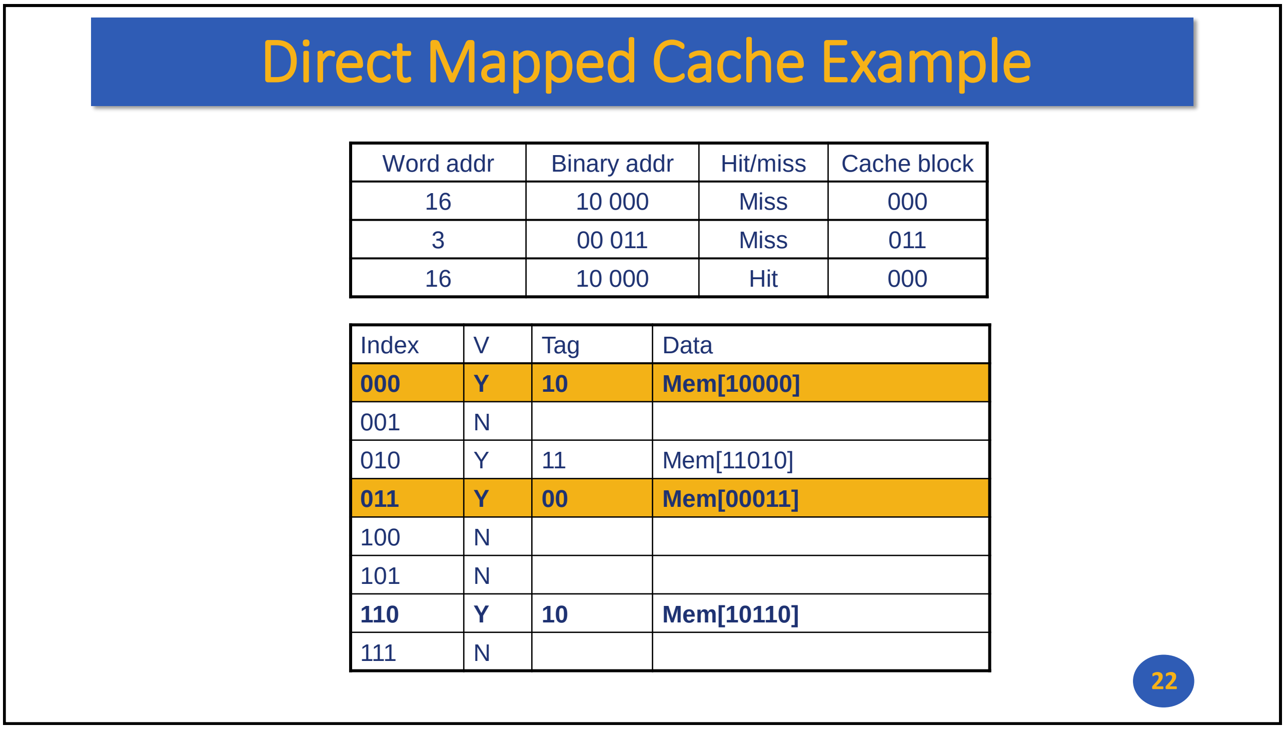
Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Direct Mapped 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		



Direct Mapped 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		

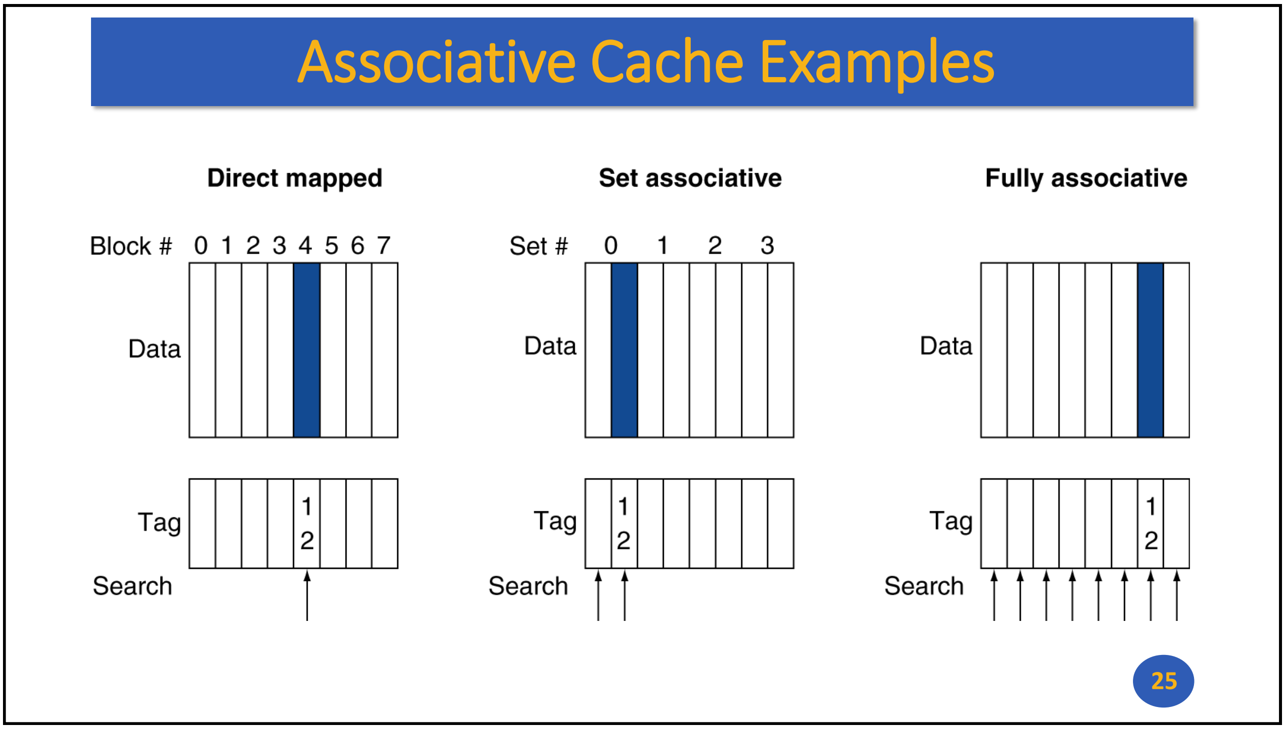
Direct Mapped Cache Example

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

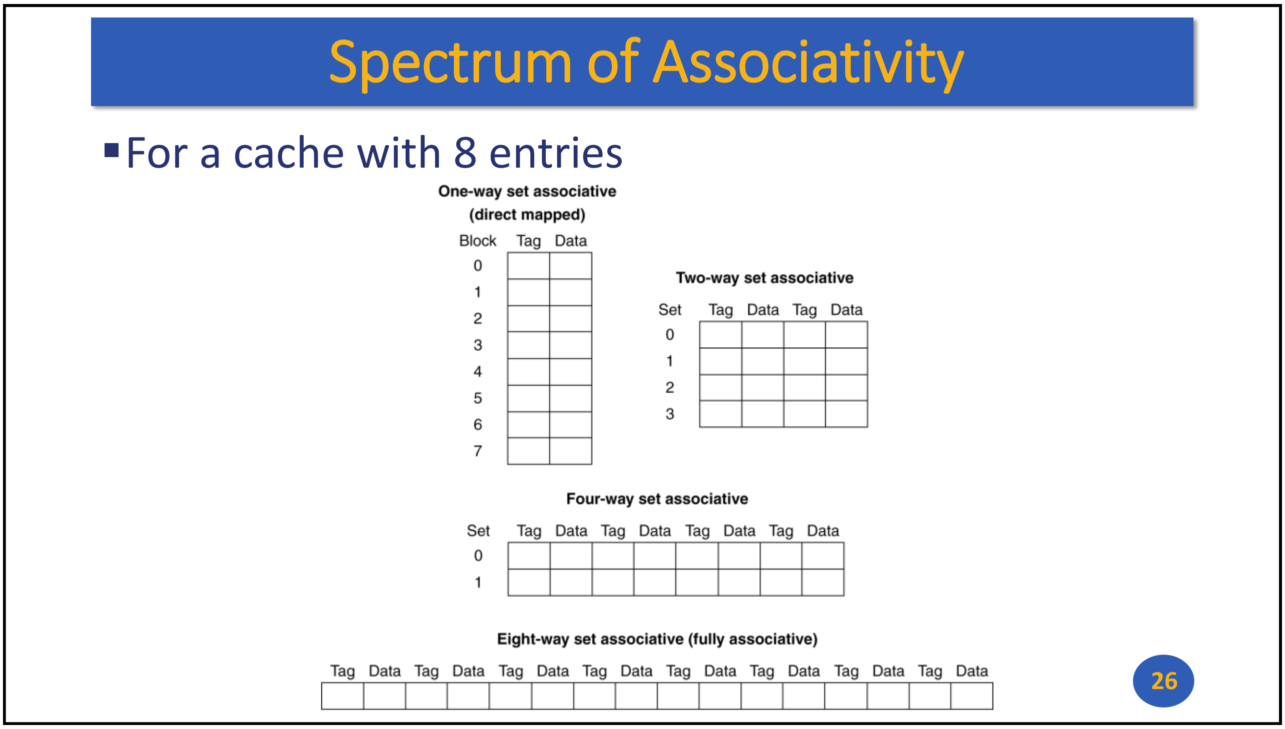
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Associative Caches

- **Fully associative**
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- ***n*-way set associative**
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)



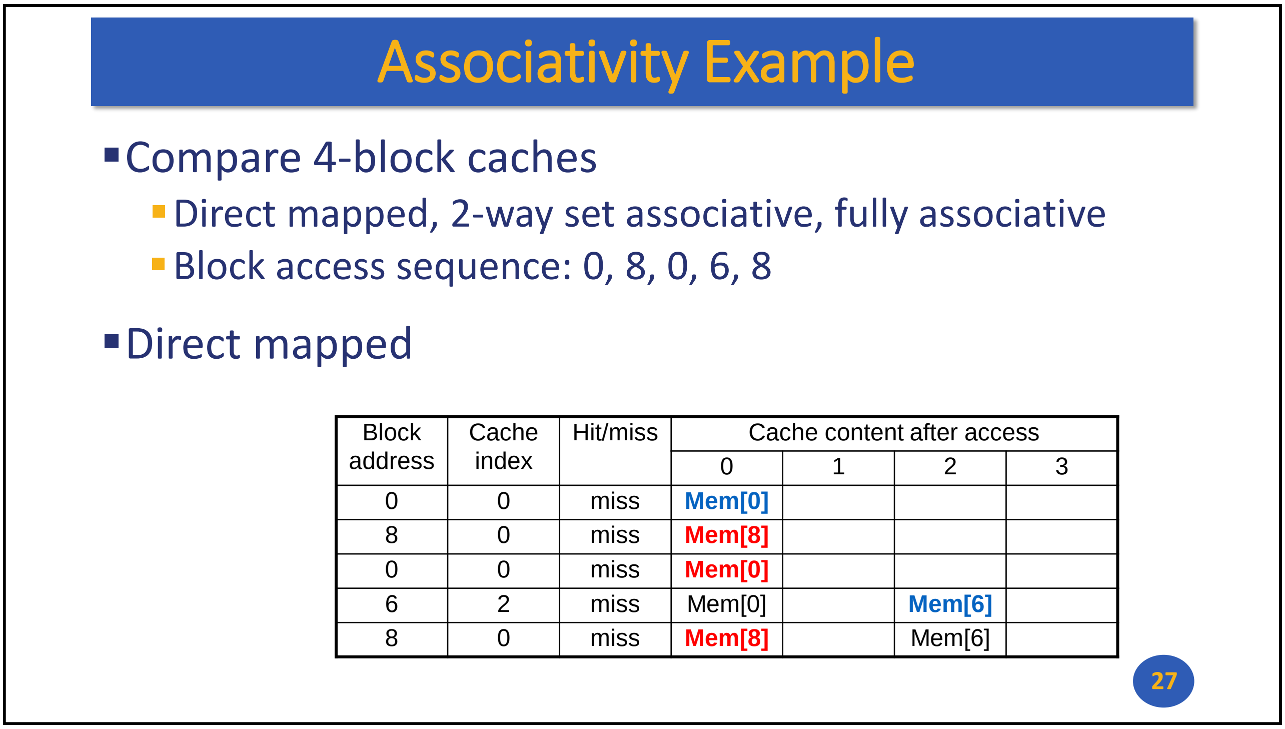
Associative Cache Examples



Spectrum of Associativity

- For a cache with 8 entries

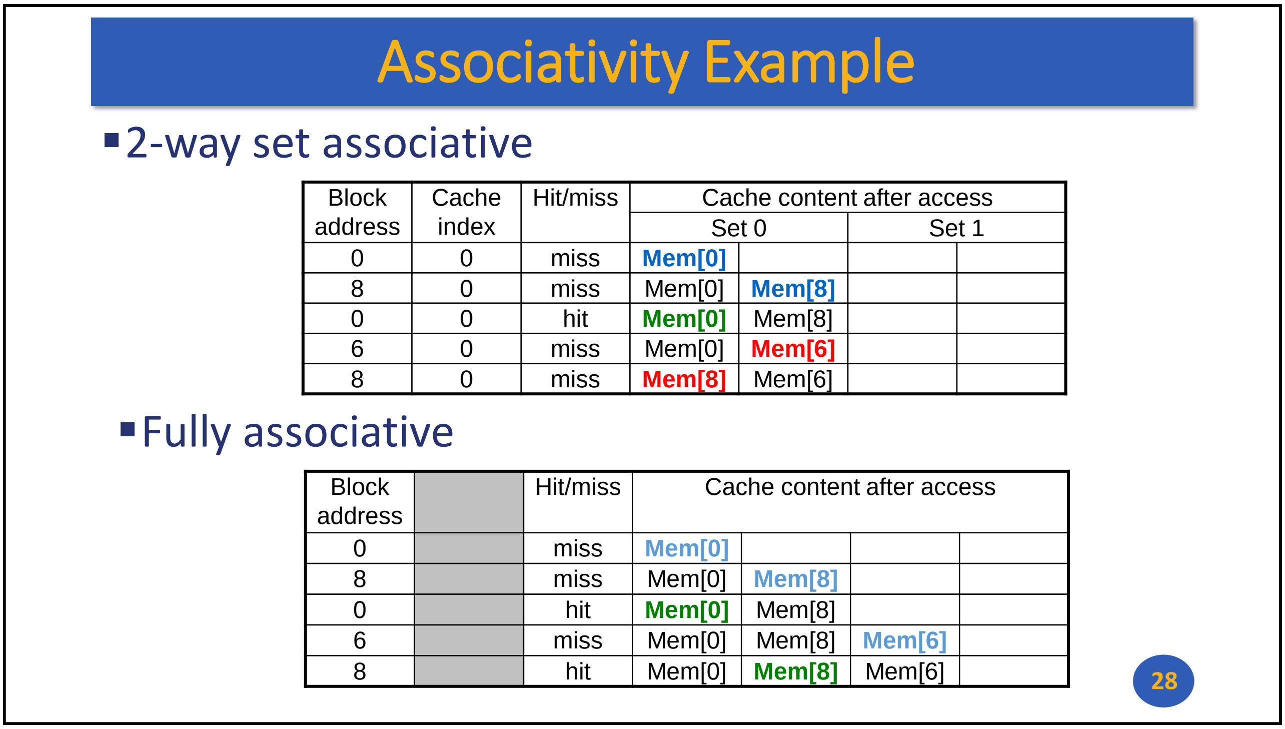
One-way set associative
(direct mapped)
Block Tag Data
0
1
2
3
4
5
6
7
Two-way set associative
Set Tag Data Tag Data
0
1
2
3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0
1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data



Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

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]	



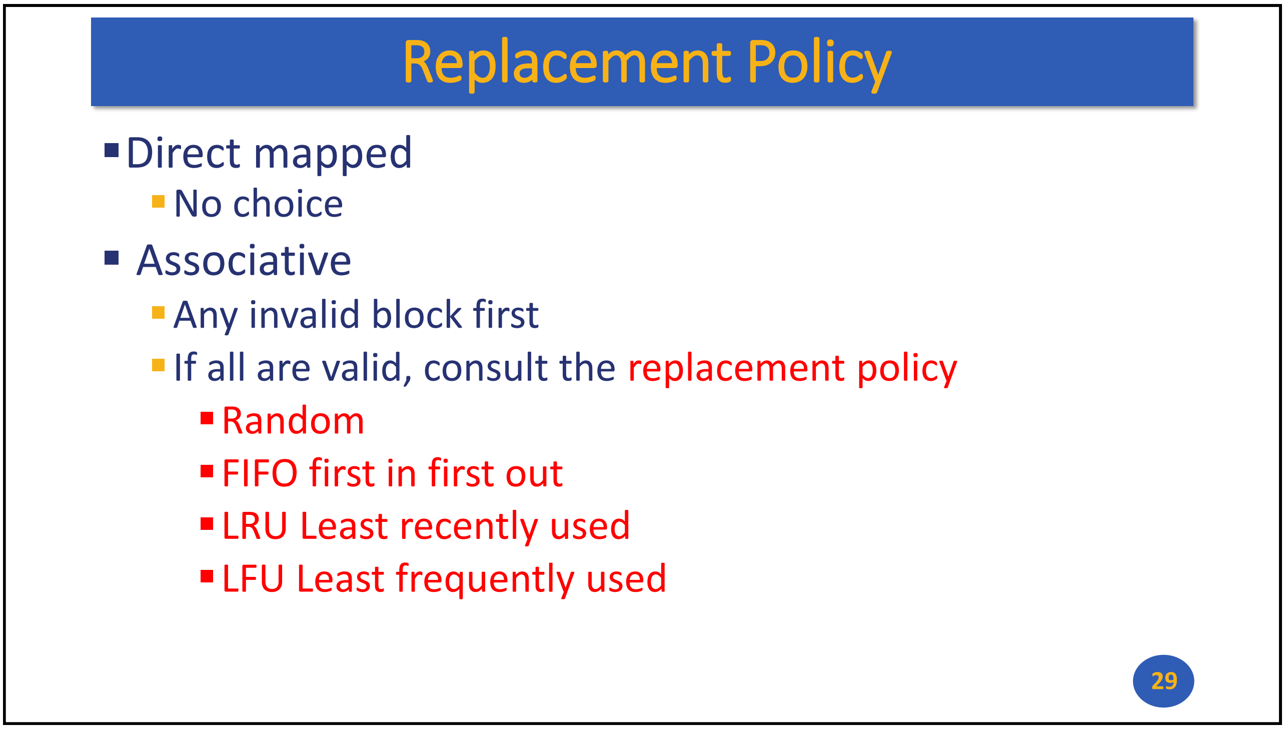
Associativity Example

- 2-way set associative

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]		

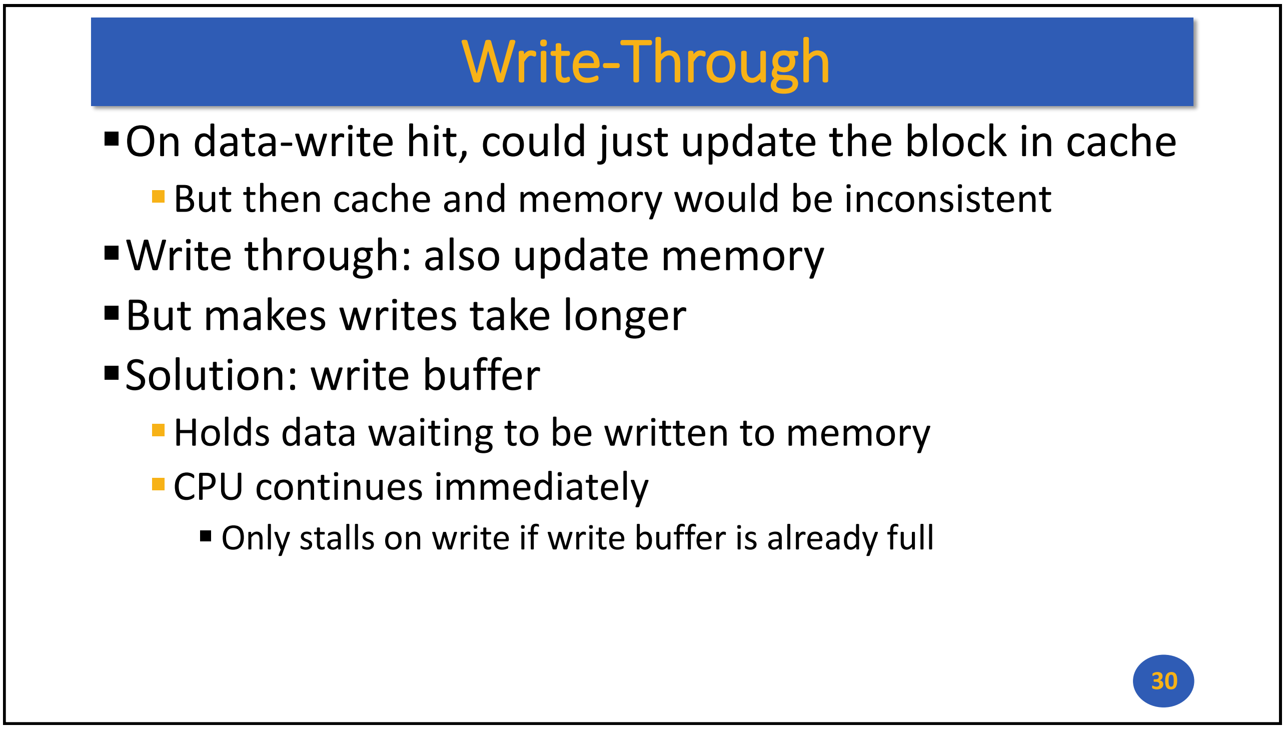
- Fully associative

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]	



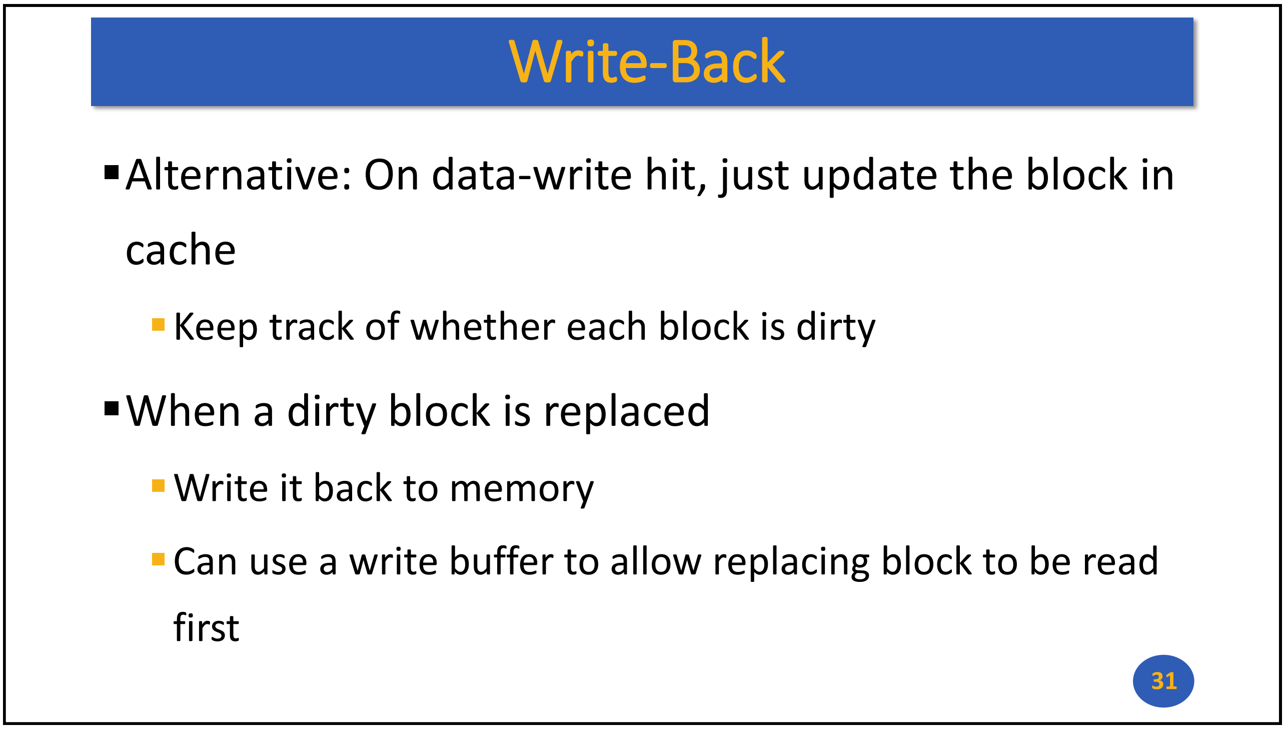
Replacement Policy

- Direct mapped
  - No choice
- Associative
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO first in first out
    - LRU Least recently used
    - LFU Least frequently used



Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full



Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first