Lecture 22

Computer Architecture
(Spring 2020)

Memory Hierarchy Design

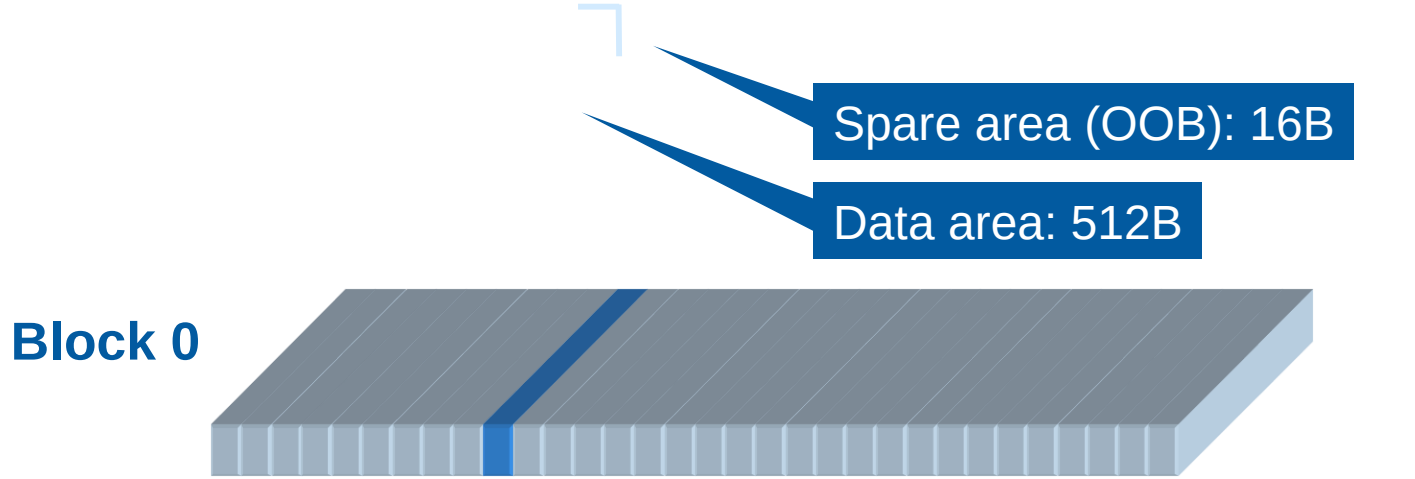
Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

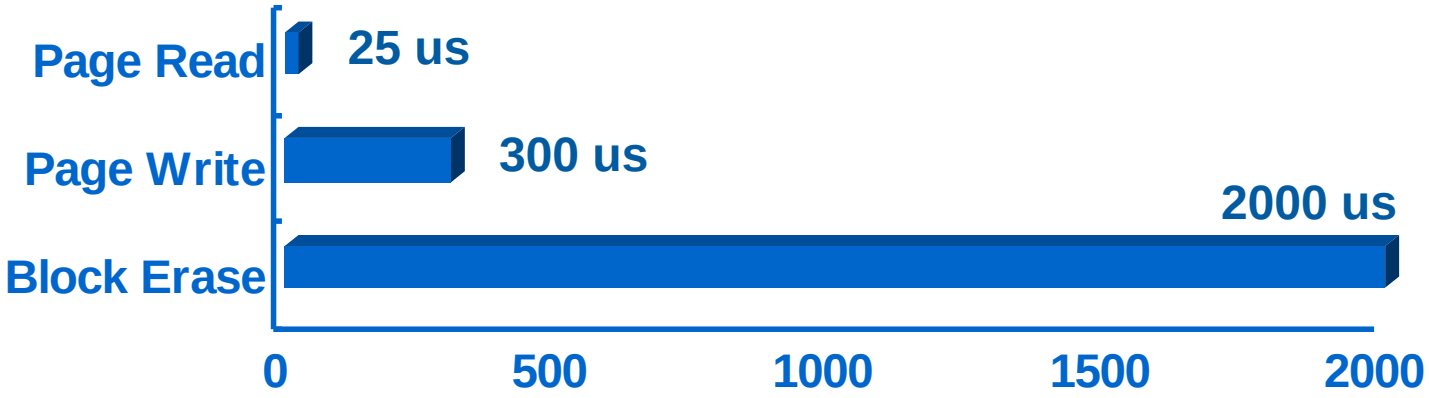
Why Flash Memory?

- Non-volatility
- Short read/write latency
- Low power consumption
- Small size and light weight
- Solid state reliability

NAND Flash Memory Organization

- **Chip → Block → Page**
  - Block = 32 / 64 pages; Page = 512B + 16B





SAMSUNG 128MB SLC NAND flash memory chip

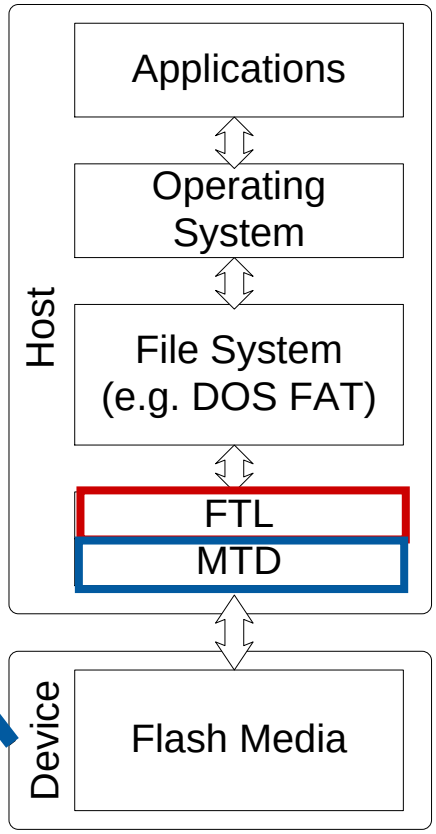
NAND Flash Memory Constraints

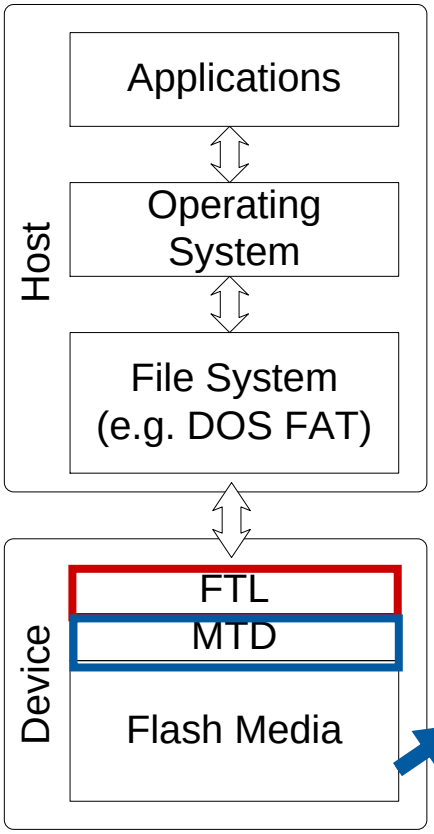
- **Out-of-place update**

- **Limited endurance: 10^4 (MLC) ~ 10^5 (SLC)**
- **Wear leveling**
- **Solution: Flash Translation Layer (FTL)**

System Architecture with FTL

SD, xD, MemoryStick, SmartMedia





CompactFlash

***FTL: Flash Translation Layer, MTD: Memory Technology Device**

DRAM vs SDRAM vs DDR SDRAM

- **Conventional DRAM**
  - **– asynchronous interface to memory controller**
  - **• every transfer involves additional synchronization overhead**
- **Synchronous DRAM**
  - **– added a clock signal so that repeated transfers would not bear that overhead**
  - **• typically have a programmable register to hold the number of bytes requested, to send many bytes over several cycles per request**
- **Double Data Rate (DDR) DRAM**
  - **– double peak bandwidth by transferring data on both clock edges**
  - **• to supply data at these high rates, DDR SDRAM activate multiple banks internally**

Clock Rate, Bandwidth, and Names of DDR DRAMs and DIMMs in 2010

Standard	Clock Rate (Mhz)	Transfers (M / sec)	DRAM name	MB/sec/ DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10644	PC10700
DDR3	800	1600	DDR3-1600	12800	PC12800
DDR4	1066-1600	2133-3200	DDR4-3200	10756-25600	PC25600

Review: Principle of Locality

- **Temporal Locality**
  - **a resource that is referenced at one point in time will be referenced again sometime in the near future**
- **Spatial Locality**
  - **the likelihood of referencing a resource is higher if a resource near it was just referenced**
- **90/10 Locality Rule of Thumb**
  - **a program spends 90% of its execution time in only 10% of its code**
    - **a consequence of how we program and we store the data in the memory**
    - **hence, it is possible to predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past**

Cache Concepts

- **The term "Cache"**
  - **– the first (from the CPU) level of the memory hierarchy**
  - **– often used to refer to any buffering technique exploiting the principle of locality**
- **Directly exploits temporal locality providing faster access to a smaller subset of the main memory which contains copy of data recently used**
- **But, all data in the cache are not necessarily data that are spatially close in the main memory…**
- **…still, when a cache miss occurs a fixed-size block of contiguous memory cells is retrieved from the main memory based on the principle of spatial locality**

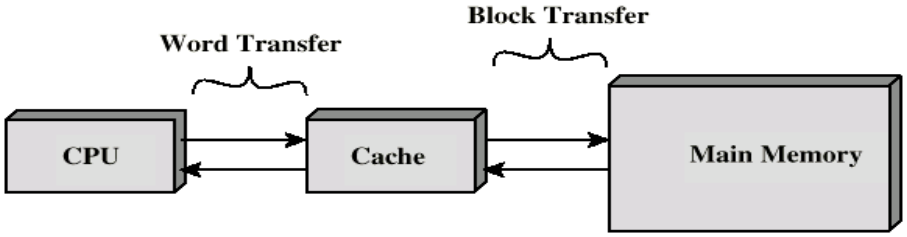
Example

```
for ( i= 0 ; i< 20 ; i++)
    for( j=0; j<10; j++)
        a[i]=a[i]*j;
```

- Spatial locality: for the outer loop, it accesses memory a[0], a[1], …,a[20] sequentially.

- Temporal locality: for the inner loop, it accesses memory a[i] repeatedly in each iteration.

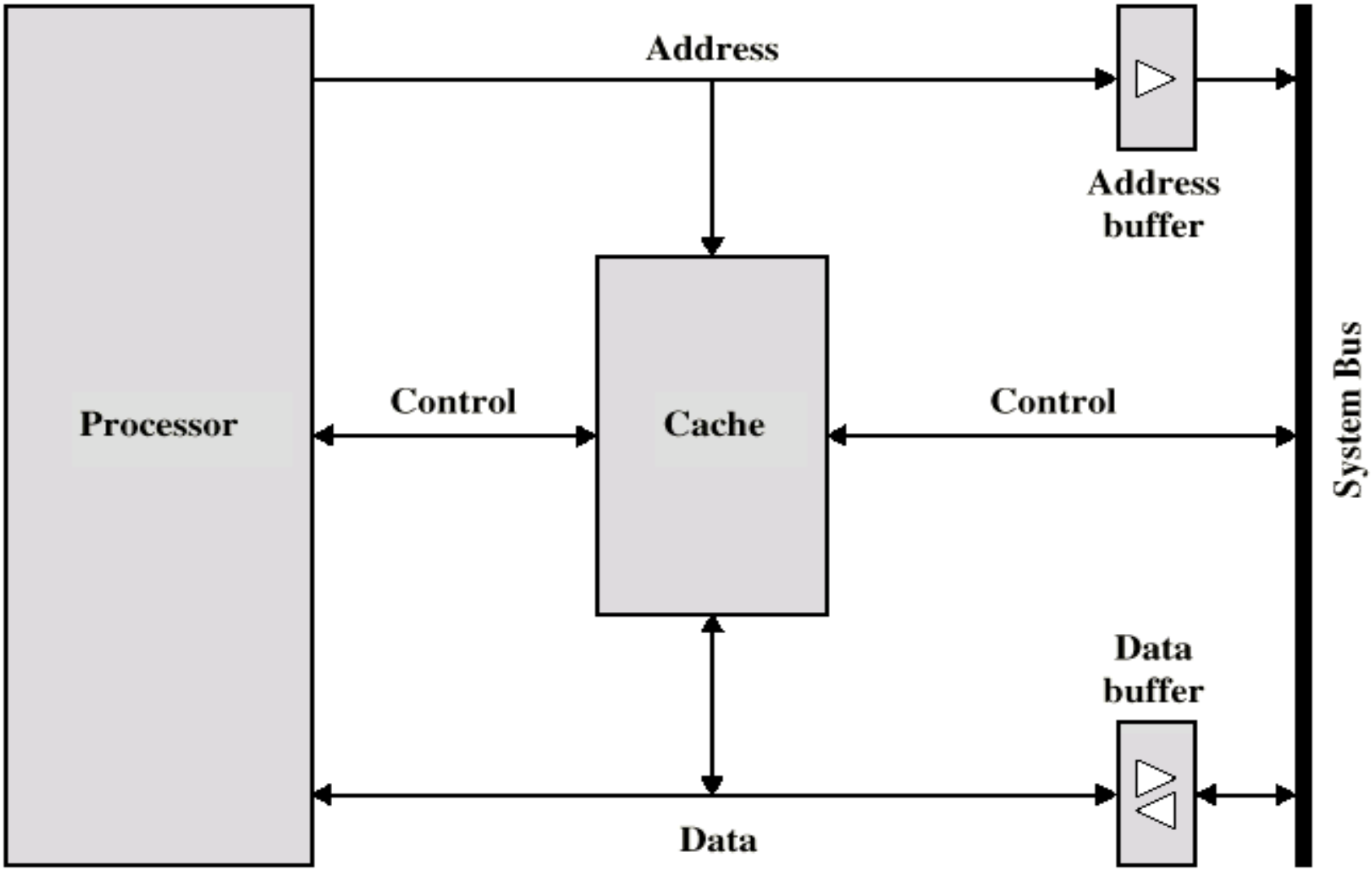
Basic Memory Hierarchies:



- When a word or an instr. is not in the cache, it will be fetched from a lower level in the hierarchy and placed in the cache before continuing.
- Multiple words, called a *block* (or *line*), are moved for efficiency reasons – spatial locality.
- Each cache block includes a *tag* to indicate which memory address it corresponds to. A key decision is where a block can be placed ?
  - The most popular scheme is *set associative*, where a *set* is a group of blocks.
    - A set with N blocks is called N-way associative
  - A block is first mapped onto a set, and then the block can be placed anywhere within that set.
  - Finding a block consists of first mapping the block address to the set and then searching the set—usually in parallel—to find the block.
  - The set is chosen by the address of the data:

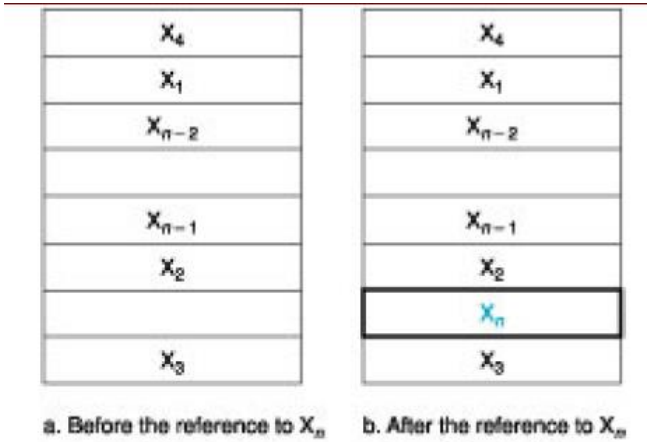
$$(\textit{Block address}) \text{ MOD } (\textit{Number of sets in cache})$$

Typical Cache Organization



Cache Concepts – cont.

- **Cache Hit**
  - **CPU find the requested data item in the cache**
- **Cache Miss**
  - **CPU doesn't find the requested data item in the cache**
- **Miss Penalty**
  - **time to replace a block in the cache (plus time to deliver data item to CPU)**
  - **time depends on both latency & bandwidth**
    - **latency determines the time to retrieve the first word**
    - **bandwidth determines the time to retrieve rest of the block**
  - **handled by hardware that stalls the memory unit (and, therefore, the whole instruction processing in case of simple single-issue uP)**



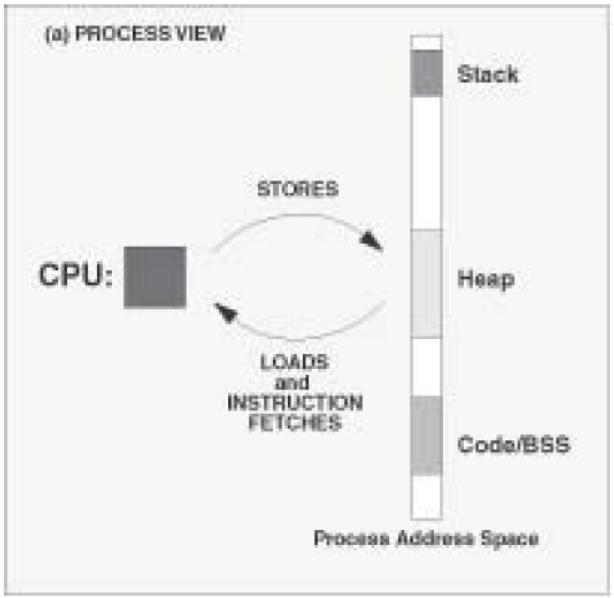
a. Before the reference to X_n b. After the reference to X_n

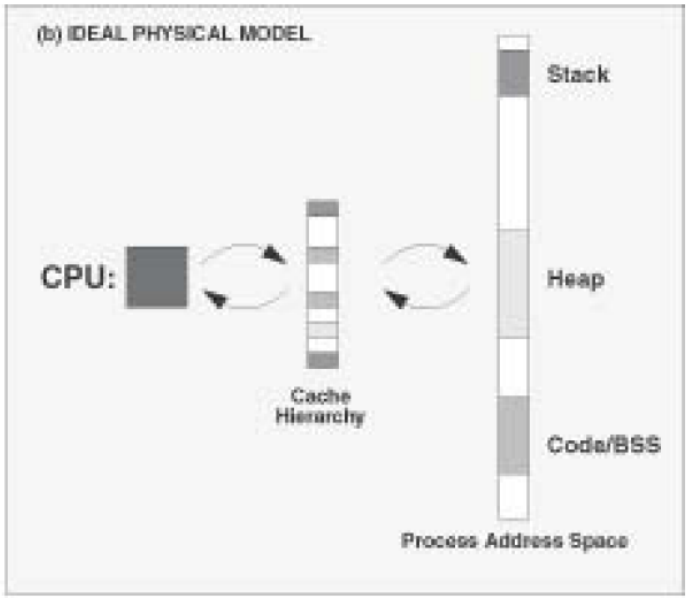
Cache : Main Memory = Main Memory : Disk

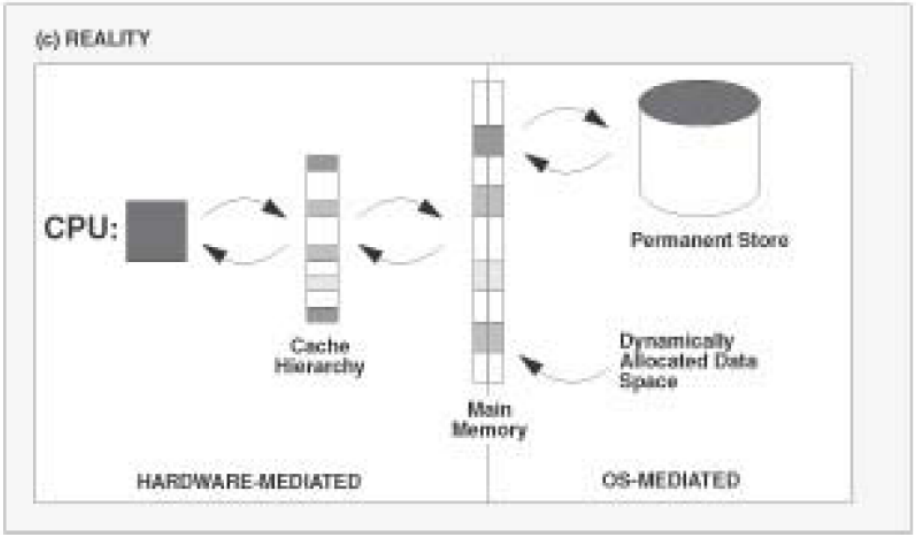
- **Virtual Memory**
  - **makes it possible to increase the amount of memory that a program can use by temporarily storing some objects on disk**
  - **program address space is divided in pages (fixed-size blocks) which reside either in cache/main memory or disk**
  - **better way to organize address space across programs**
    - **necessary protection scheme to control page access**
  - **when the CPU references an item within a page that is not present in cache/main memory a page fault occurs**
    - **the entire page is moved from the disk to main memory**
  - **page faults have long penalty time**
    - **handled in SW without stalling the CPU, which switches to other tasks**

Caching the Address Space

- Programs today are written to run on no particular HW configuration
- Processes execute in imaginary address spaces that are mapped onto the memory system (including DRAM and disk) by the OS
- Every HW memory structure between the CPU and the permanent store is a cache for instruction & data in the process's address space

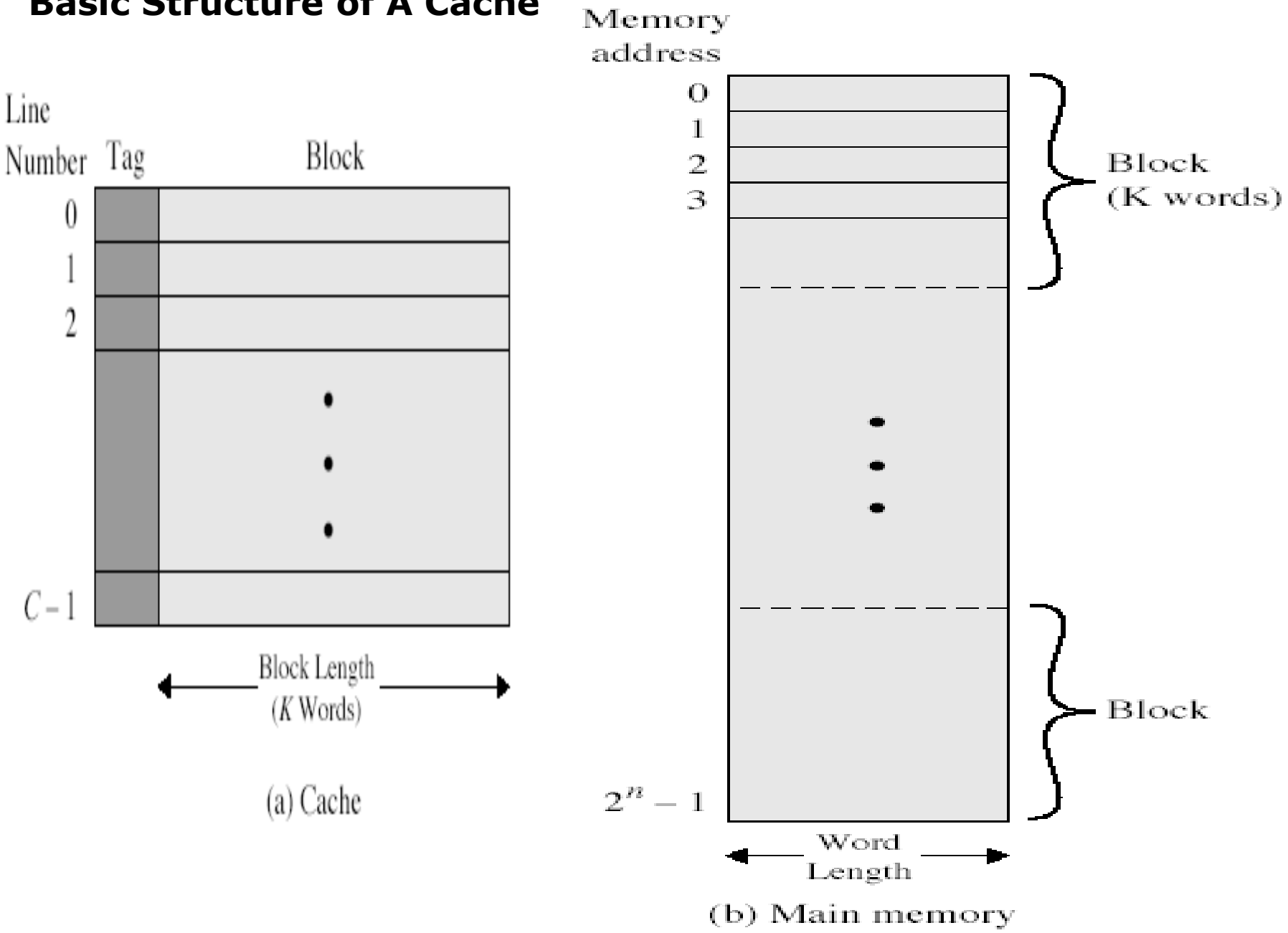






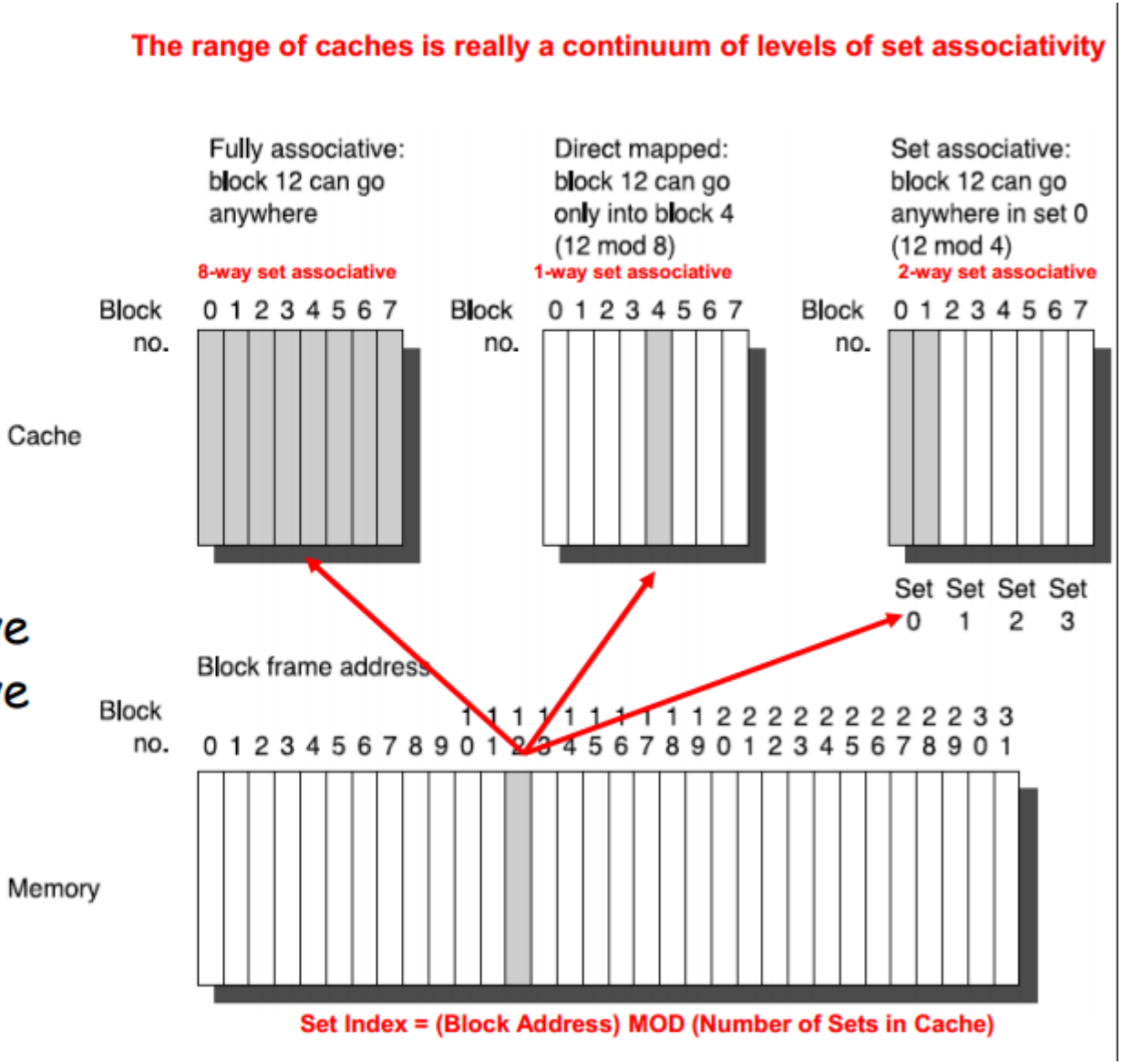
Source: B. Jacob et al. "Memory Systems"

Basic Structure of A Cache



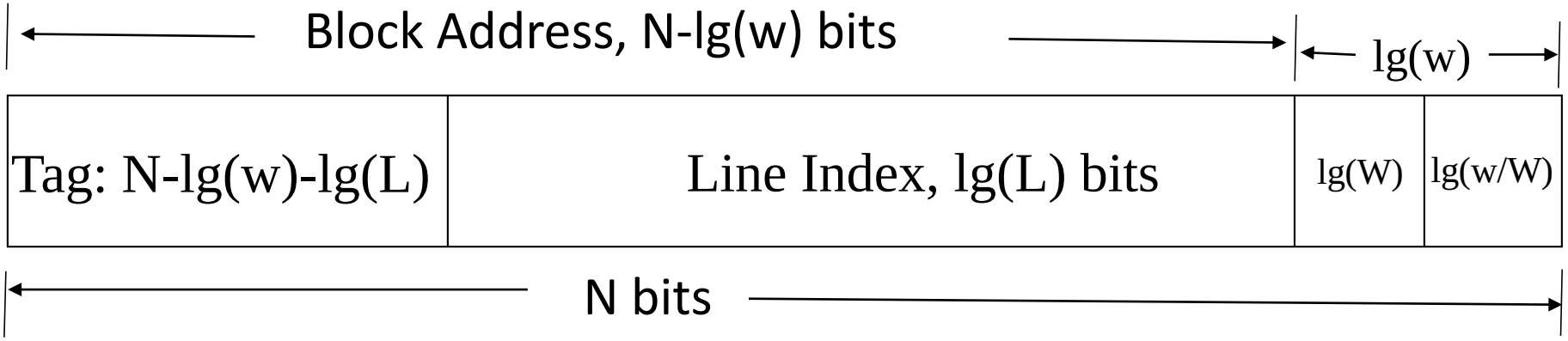
Cache Schemes: Placing a Memory Block into a Cache Block Frame

- **Block**
  - unit of memory transferred across hierarchy levels
- **Set**
  - a group of blocks
- **Modern processors**
  - direct map
  - 2-way set associative
  - 4-way set associative
- **Modern memories**
  - millions of blocks
- **Modern caches**
  - thousands of block frames



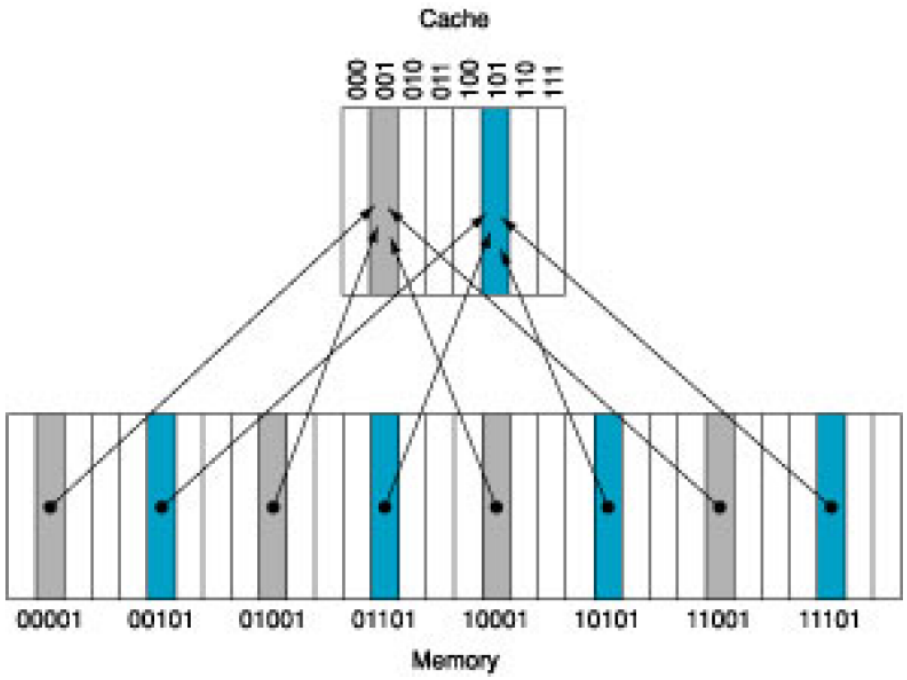
Direct Mapping

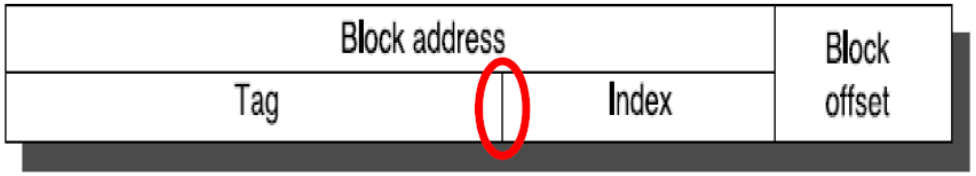
- Each block of main memory maps to only one cache line.
- Assume total N-bit memory address. It can address 2^N bytes.
- Assume each block has w bytes. Upon a miss, a block (total w bytes of data) is fetched from memory.
  - Alternatively, you can say each block has W words, w = 4W.
  - So byte address (or offset) within a block is lg(w) bits, and word address (or offset) is lg(W) bits.
  - So the number of blocks in main memory is $2^N/w$, hence the block address is $\lg(2^N/w) = N-\lg(w)$
- Assume the cache can host L lines/blocks. So the line's index is lg(L) bits.
- When we place a block to a cache line, the block address is split into two parts:
  - The LSB of lg(L) bits in block address tells the cache line the block maps to.
  - The rest MSB bits, N-lg(w)-lg(L) in block address becomes a tag.



Example: Direct Mapped Cache with 8 Block Frames

- Each memory block is mapped to one cache entry
  - cache index = (block address) mod (# of cache blocks)
  - e.g., with 8 blocks, 3 low-order address bits are sufficient
    - Log2 (8) = 3
- Is a block present in cache?
  - must check cache *block tag*
    - upper bit of block address
- Block offset
  - addresses bytes in a block
    - block==word ⇒ offset =2 bits
- How do we know if data in a block is valid?
  - add valid bit to each entry





The tag index boundary moves to the right as we increase associativity (no index field in fully associative caches)

Example: Measuring Cache Size

- How many total bits are required for a direct-mapped cache with 16KB of data and 4-word block frames assuming a 32-bit address?
- 16KB of data = 4K words = 2^{12} words
- Block Size of 4 ($=2^{2}$) words $\Rightarrow$ 2^{10} blocks

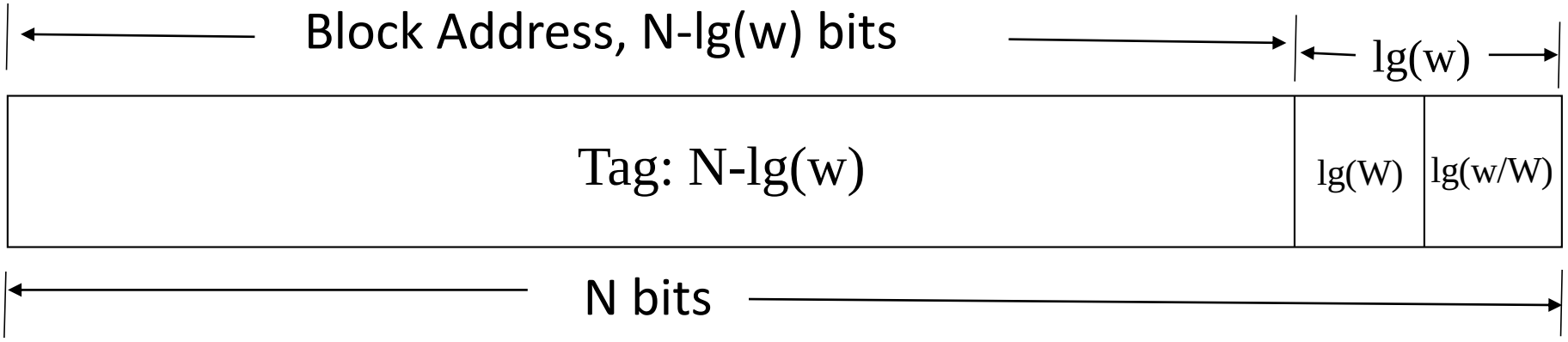
- # Bits in a Tag = 32 - (10 + 2 + 2) = 18
- # Bits in a block = # Tag Bits + # Data Bits + Valid bit
- # Bits in a block = 18 + (4 * 32) + 1 = 147
- Cache Size= # Blocks x #Bits in a block= 2^{10} x 147=147Kbits
- Cache Overhead = 147Kbits / 16KB = 147 / 128 = 1.15

Direct Mapping pros & cons

- Simple, and Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

Fully Associative Mapping

- Each block of main memory maps to any cache line. CPU chooses which line to map.
- Assume total N-bit memory address, can address 2^N bytes.
- Assume each block has w bytes. Upon a miss, a block (total w bytes of data) is fetched from memory.
  - Alternatively, you can say each block has W words, w = 4W.
  - So byte address (or offset) within a block is lg(w) bits, and word address (or offset) is lg(W) bits.
  - So the number of blocks in main memory is $2^N/w$, hence the block address is $\lg(2^N/w) = N-\lg(w)$
- When we place a block to a cache line, the whole block address is used as its tag.
  - The whole block address, which is N-lg(w) bits, becomes the tag.

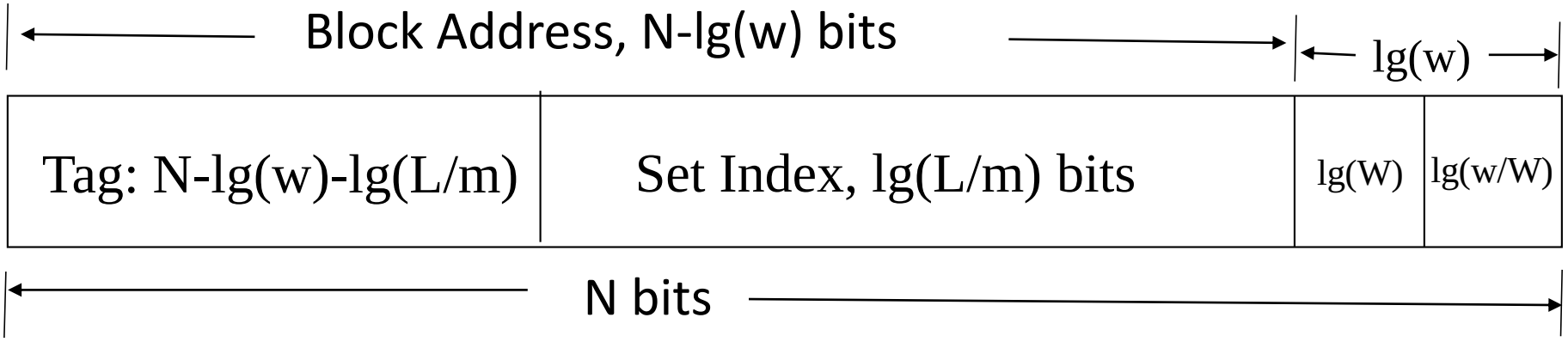


Fully Associative Mapping pros and cons

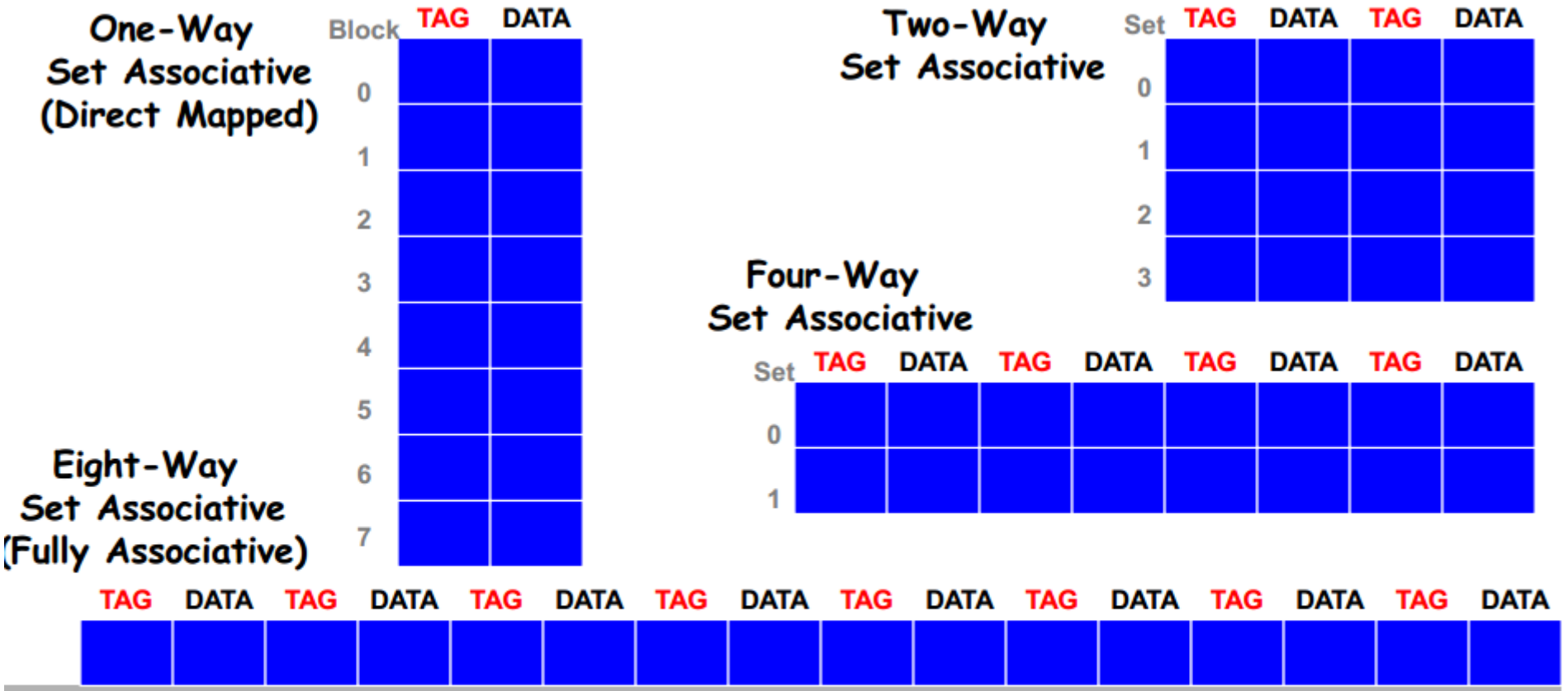
- Full flexibility on determining the location in cache.
- Expensive: Every line's tag is examined for a match; Cache searching gets slow!

Set Associative Cache

- Cache is organized into sets where each set has 2, 4, 8 or more blocks.
  - Each block is mapped to a set, and can be placed anywhere in that set.
  - The cache using m blocks per set is called m-way cache.
  - The number of sets is: L/m. Hence the set index is lg(L)-lg(m);
  - It is a trade-off between Direct Map (1-way) and Full Associative Map (All-way).
- When we place a block to a cache line, the block address is split into two parts:
  - The LSB of lg(L)-lg(m) bits in block address tells the cache set the block maps to.
  - The rest MSB bits, N-lg(w)-lg(L)-lg(m) in block address becomes its tag.



Cache Schemes: Storage Space Organization



- Cache total size (in blocks) = #sets x associativity
  - increasing the associativity decreases the number of sets, while increasing the number of blocks per set
    - 8-way set-associative cache with 8 blocks ≡ fully associative cache

Example: Mapping Function

Example Architecture we will use:

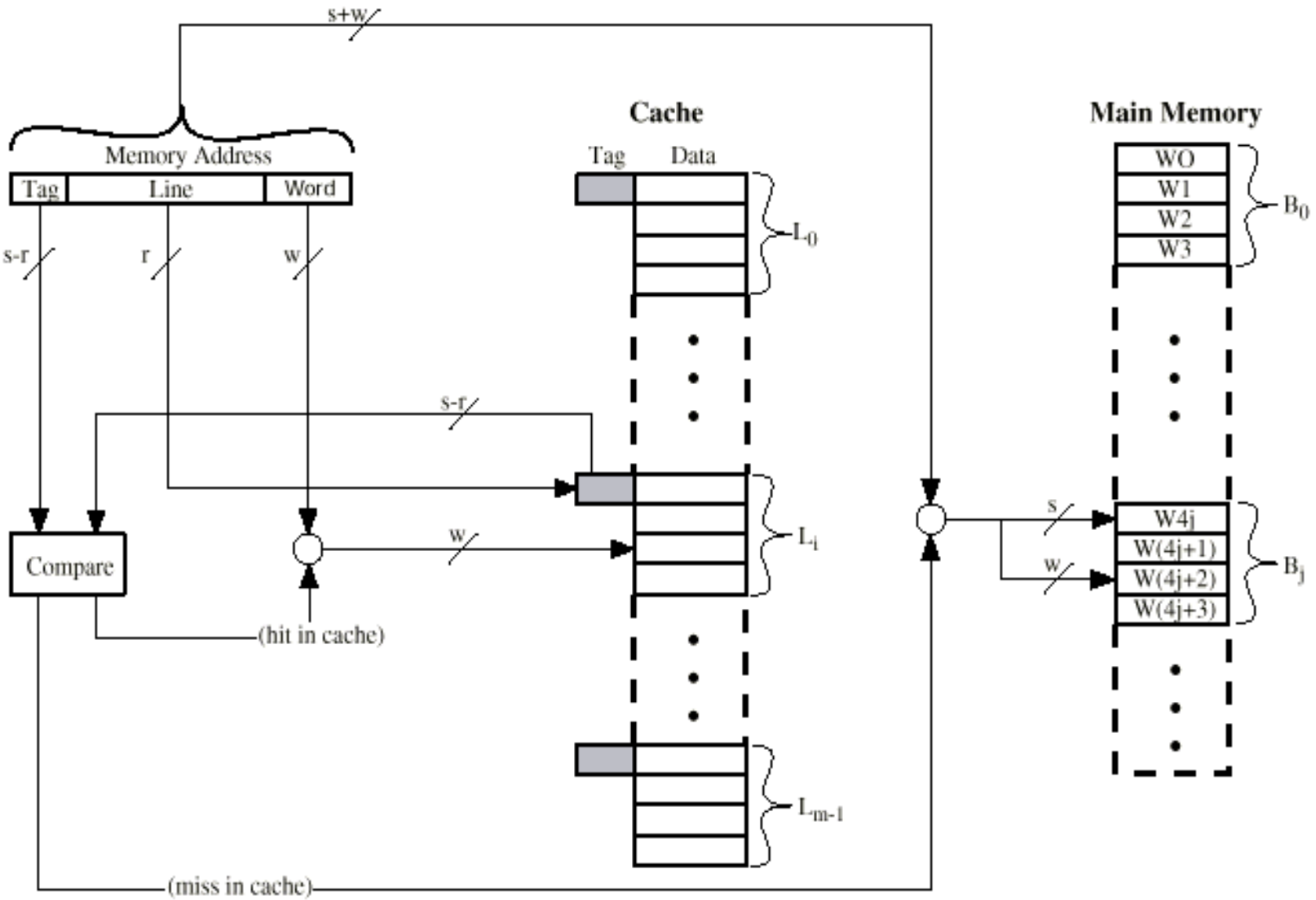
- **Cache of 64kByte**
- **Cache block of 4 bytes**
  - i.e. cache is 16k (2^{14}) lines of 4 bytes
- **16MBytes main memory**
- **24 bit address (2^{24}=16M)**

Direct Mapping Address Structure

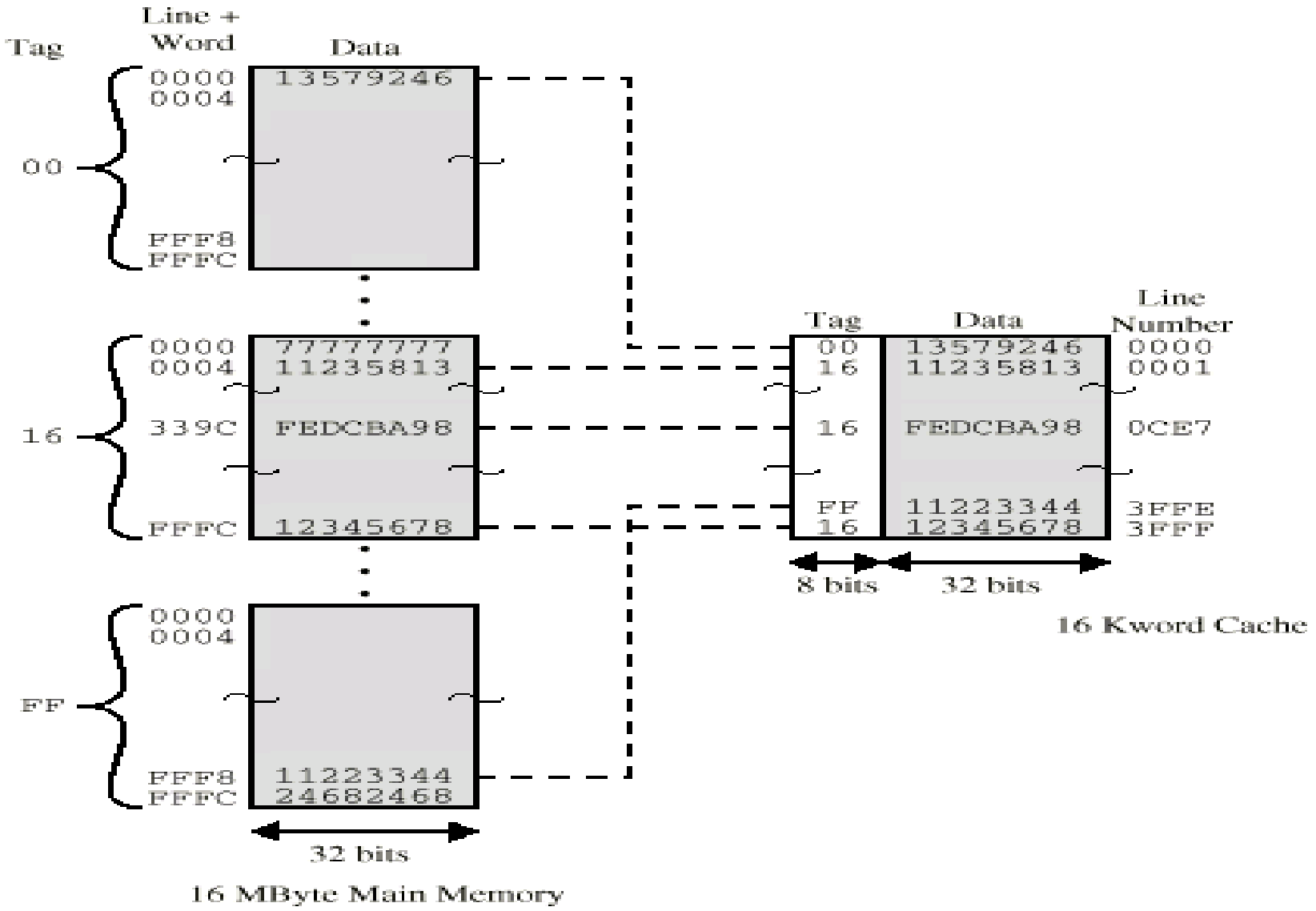
Tag s-r	Line or Slot r	Word w
8	14	2

- **24 bit address**
- **2 bit word identifier (4 byte block)**
- **22 bit block identifier**
  - 8 bit tag (=22-14)
  - 14 bit slot or line
- **No two blocks in the same line have the same Tag field**
- **Check contents of cache by finding line and checking Tag**

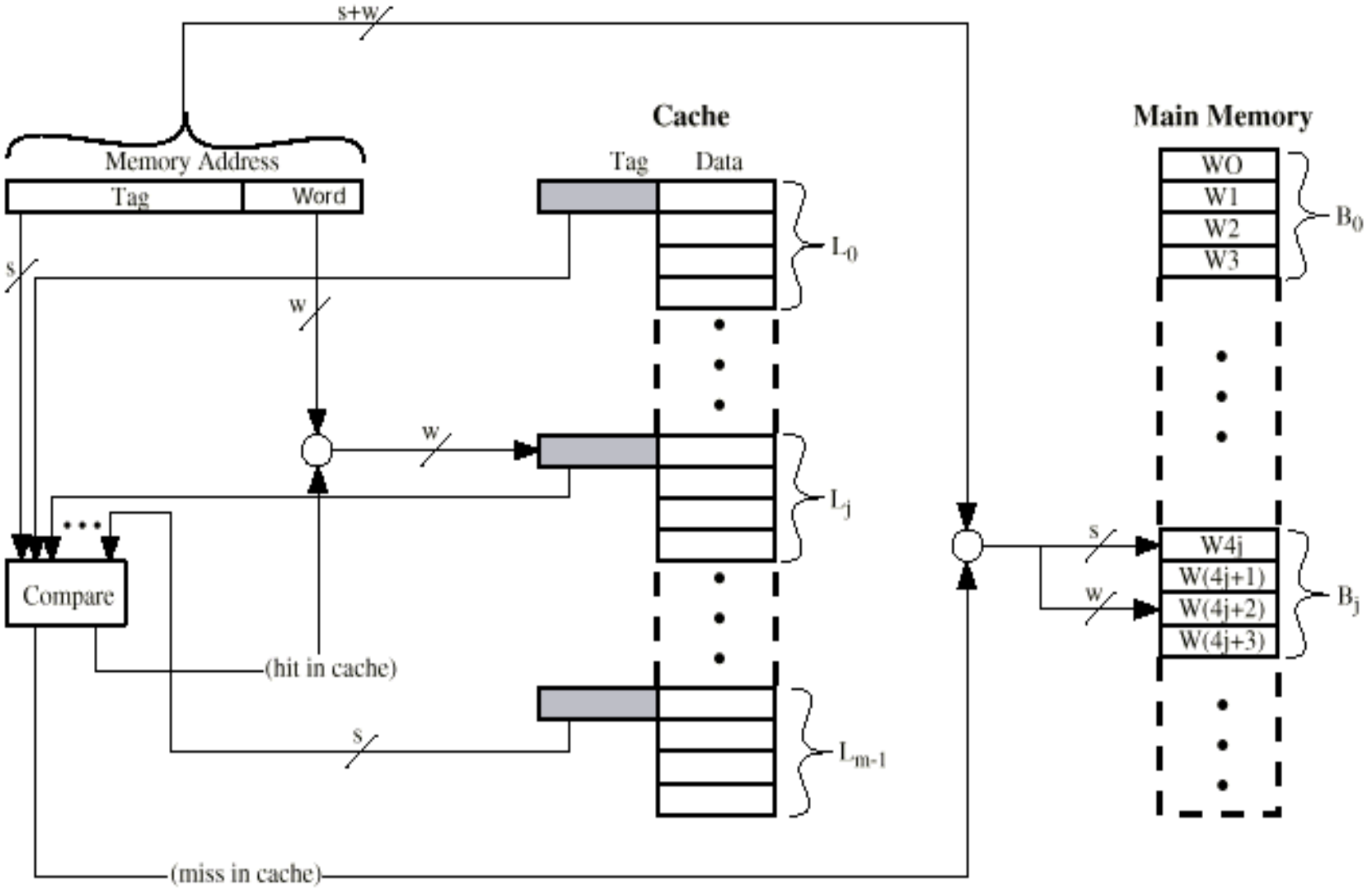
Direct Mapping Cache Organization



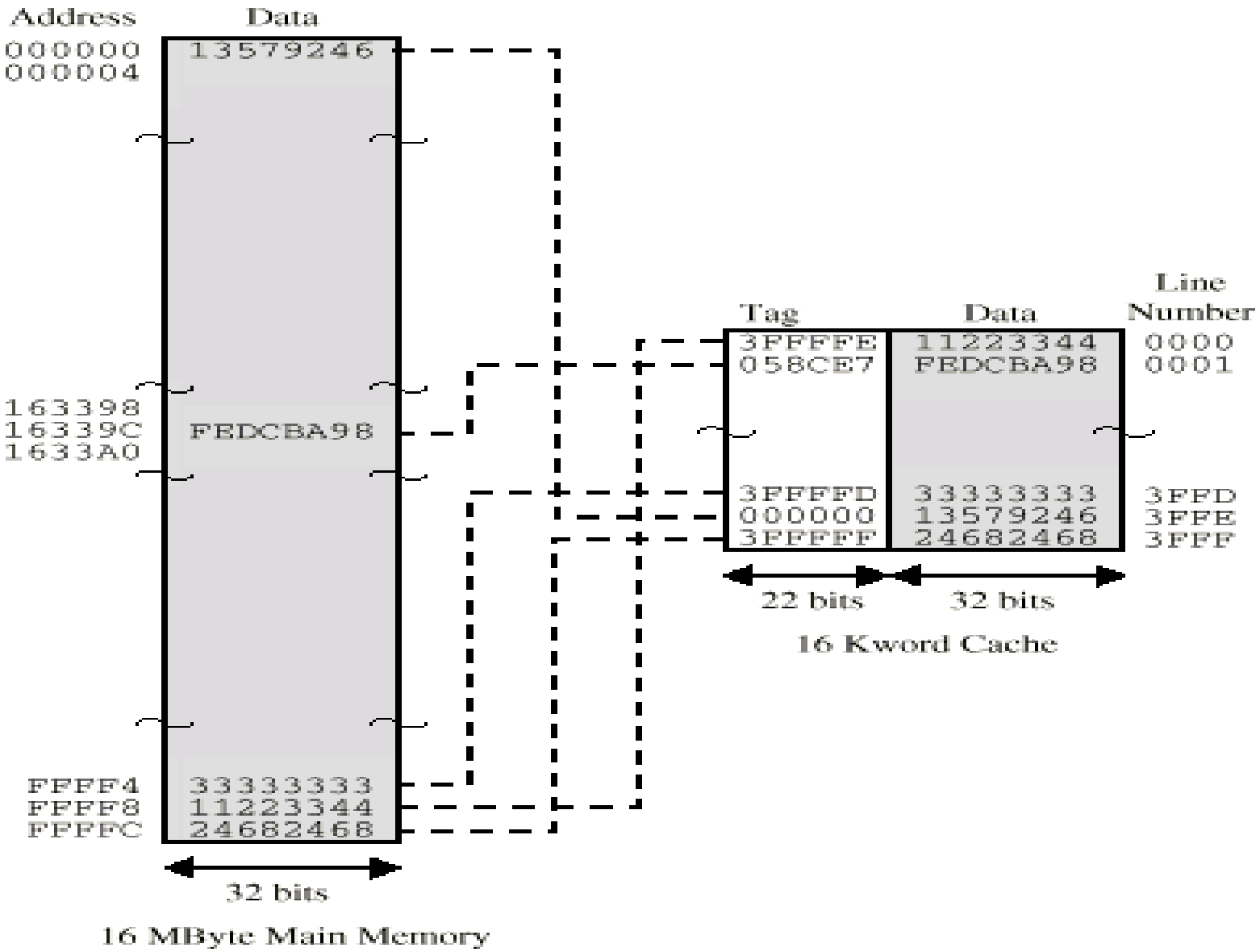
Direct Mapping Example



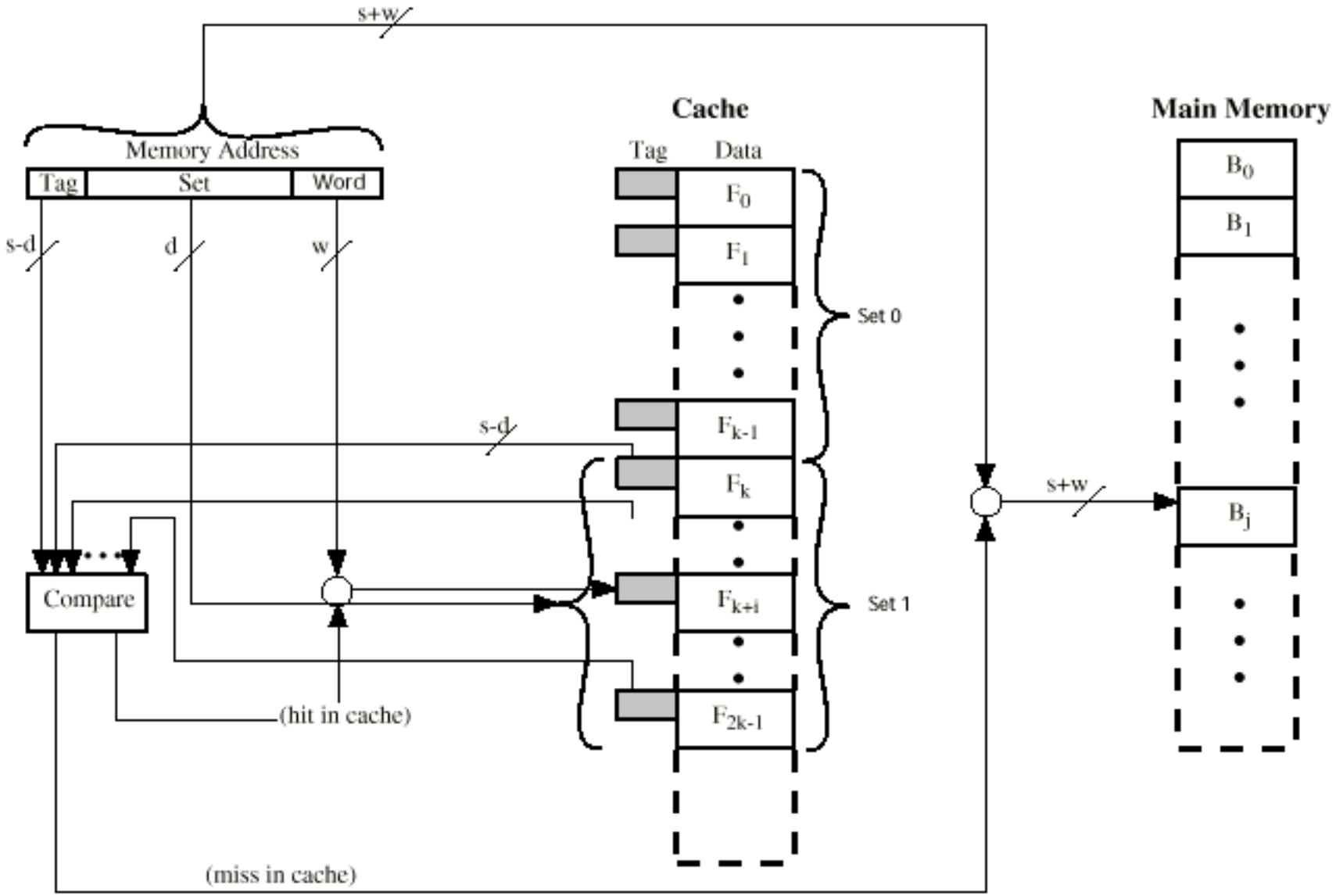
Fully Associative Cache Organization



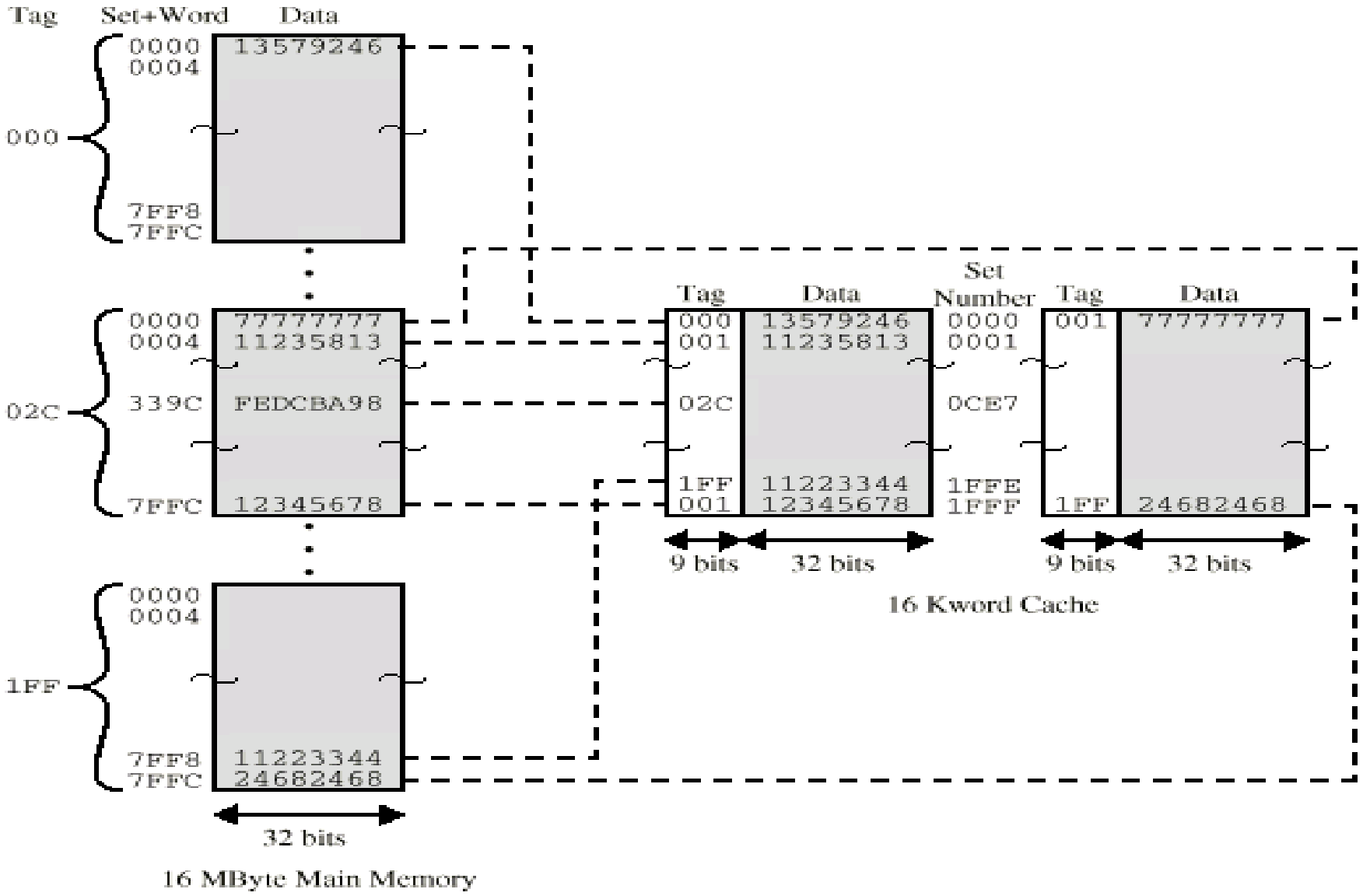
Associative Mapping Example



Two Way Set Associative Cache Organization

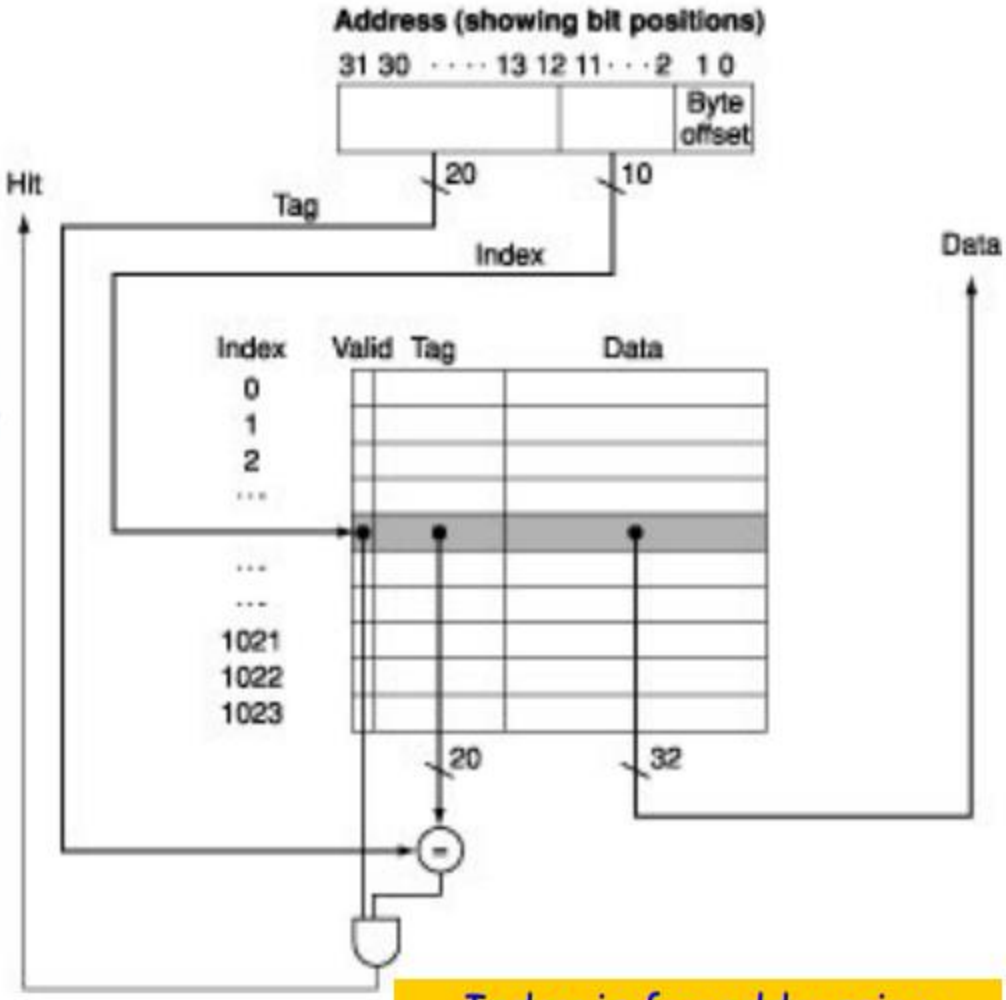


Two Way Set Associative Mapping Example



Ex: Direct Mapped with 1024 Blocks Frames and Block Size of 1 Word for MIPS-32

- **Block Offset**
  - is just a byte offset because each block of this cache contains 1 word
- **Byte Offset**
  - least significant 2 bits because in MIPS-32 memory words are aligned to multiples of 4 bytes
- **Block Index**
  - 10 low-order address bits because this cache has 1024 block frames
- **Block Tag**
  - remaining 20 address bits in order to check that the address of the requested word matches the cache entry



Index is for addressing
Tag is for checking/searching

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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

- **Assumption**
  - **–8 block frames**
  - **– block size = 1 word**
  - **– main memory of 32 words**
  - **•toy example**
  - **– we consider ten subsequent accesses to memory**

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2			
3			
4			
5			
6			
7			
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3			
4			
5			
6			
7			
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4			
5			
6			
7			
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5			
6			
7			
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5	10000	16	miss
6			
7			
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5	10000	16	miss
6	00011	3	miss
7			
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5	10000	16	miss
6	00011	3	miss
7	10000	16	hit
8			
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5	10000	16	miss
6	00011	3	miss
7	10000	16	hit
8	10010	18	miss
9			
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5	10000	16	miss
6	00011	3	miss
7	10000	16	hit
8	10010	18	miss
9	11010	26	miss
10			

Example: Accessing a Direct Mapped Cache with 8 Blocks and Block Size of 1 Word

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		

cycle	Memory Address	address in decimal	Cache Event
1	10110	22	miss
2	11010	26	miss
3	11010	26	hit
4	10110	22	hit
5	10000	16	miss
6	00011	3	miss
7	10000	16	hit
8	10010	18	miss
9	11010	26	miss
10	11010	26	hit