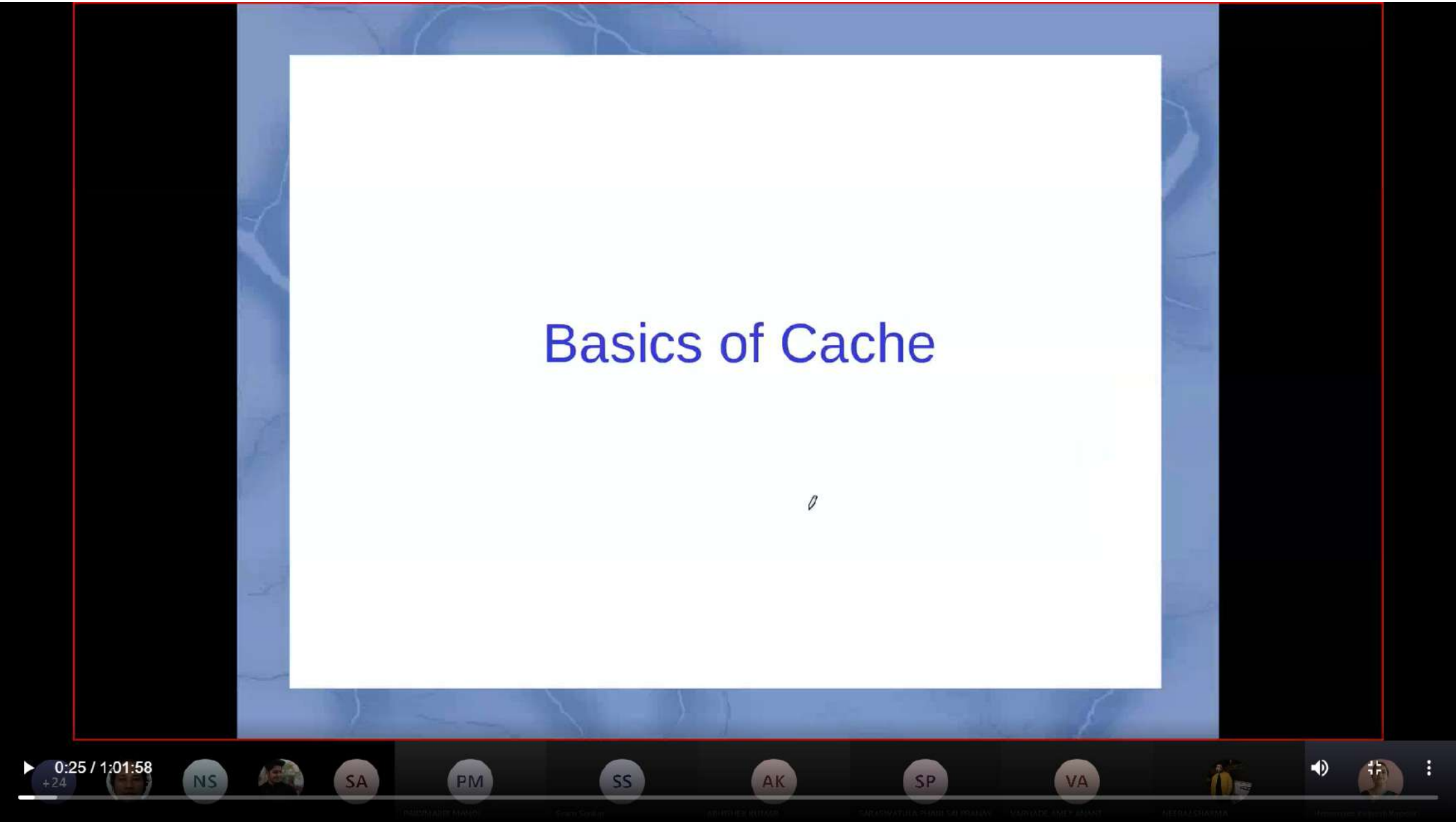

Basics of Cache
0:25 / 1:01:58
+24
NS
SA
PM
SS
AK
SP
VA

# The concept

- Analogy: to write a term paper by referring books in library
  - You are sitting at a desk
  - Bring most needed books from the shelf
  - Fetch newer books when needed
  - Bring more than one closeby book when you get up
- Temporal locality
- Spatial locality
- Applying to memory hierarchy
  - Gives illusion that a large memory can be accessed as fast as a small memory

# Principle of Locality

- States that programs access a relatively small portion of address space at any instant of time
  - Same as you accessed a very small portion of library's collection
- Temporal/Spatial in programs
  - Loops, array, instr exe in sequential manner
- Memory hierarchy
  - Multiple levels with different speed + size
  - Faster mem = more expensive per bit than slower mem => therefore small in size
  - Faster = closer to processor
- Goal = present the user with as much memory as available in the cheapest technology, while providing access at the speed offered by fastest memory

# Data hierarchy

- Level closer to processor is subset of any level further away
- All data stored in lowest level
- Min. Unit of info/data that can be present/absent is called a block or line
  - Analogy = 1-book in library
- Terms: cache miss, hit, miss-rate, hit rate, hit time, miss penalty
  - Miss penalty = analogy is time to access book on desk is fast than going to shelf

# Memory : Performance

- Memory is critical to performance
- Concepts used to build memory systems affect other aspects
  - How OS manages memory and I/O
  - How compiler generates code
  - How applications use computer
- BECAUSE?
  - Programs spend most of their time accessing memory
  - Memory systems are major factor determining performance
- Therefore, computer designers devote great deal of attention to memory systems
  - And make sophisticated mechanisms to improve memory performance

# What is Cache?

- Cache
  - Is the name given to the higest or first-level of the memory hierarchy encoutered once the address leaves the processor
  - Small and fast accesses between processor and main memory
  - Blocks brought from main memory to cache
- Cache-hit
- Cache-miss
- Temporal locality
  - Same item may be needed again
- Spatial locality
  - Nearby items will be needed in near future

# Q1: Block Placement

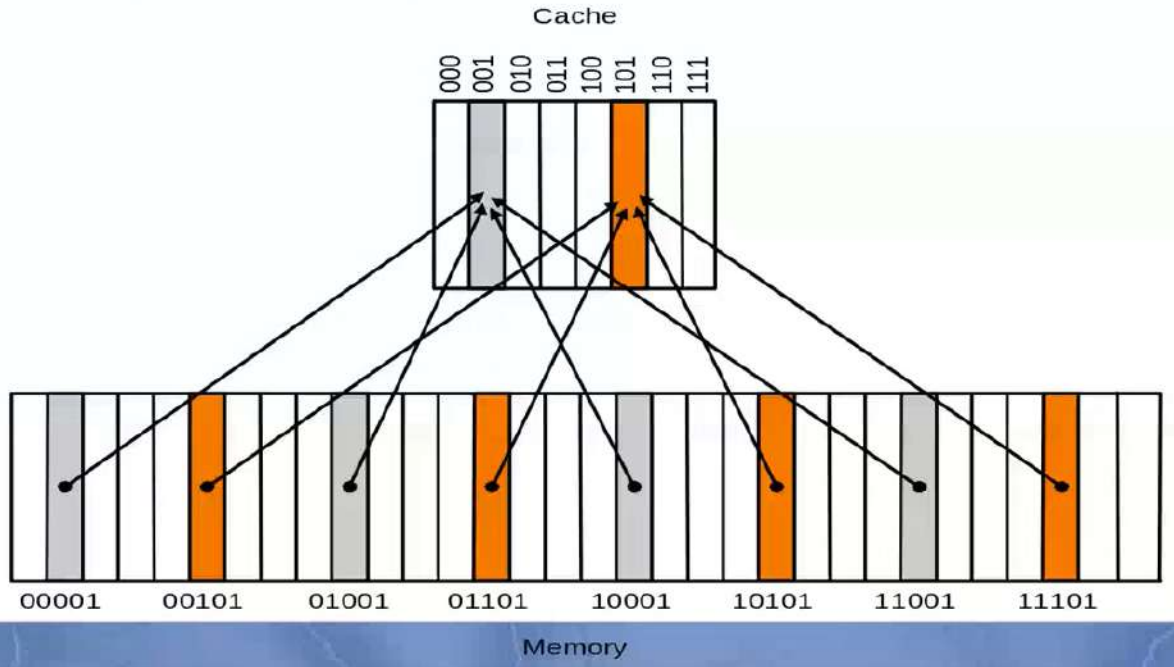
- Where can block be placed in cache?
  - In one predetermined place - direct-mapped
    - » Use part of address to calculate block location in cache
    - » Compare cache block with tag to check if block present
  - Anywhere in cache - fully associative
    - » Compare tag to every block in cache
  - In a limited set of places - set-associative
    - » Use portion of address to calculate set (like direct-mapped)
    - » Place in any block in the set
    - » Compare tag to every block in set
    - » Hybrid of direct mapped and fully associative

# Cache types

- Direct-mapped
  - Memory location maps to single specific cache line (block)
  - What if two locations map to same line (block)?
    - » Conflict, forces a miss
- Set-associative
  - Memory location maps to a *set* containing several blocks.
  - Each block still has tag and data, and sets can have 2,4,8,etc. blocks. Blocks/set = associativity
  - Why? Resolves conflicts in direct-mapped caches.
    - » If two locations map to same set, one could be stored in first block of the set, and another in second block of the set.
- Fully-associative
  - Cache only has one set. All memory locations map to this set.
  - This one set has all the blocks, and a given location could be in any of these blocks
  - No conflict misses, but costly. Only used in very small caches.

# Direct mapped cache

- **Mapping**
  - **Cache address is Memory address modulo the number of blocks in the cache**
  - **Find a cache location:**
    - (Block address) modulo (#Blocks in cache)



# Example: Accessing A Direct-Mapped Cache

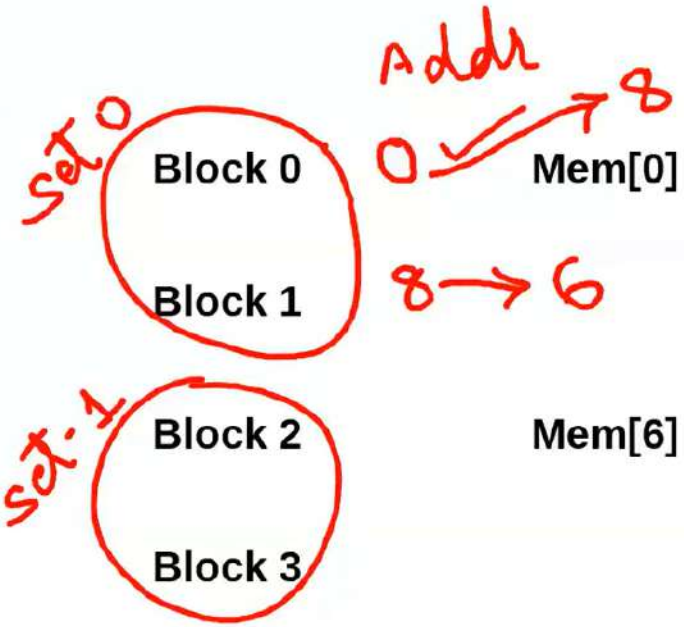
- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

| Mem Block | DM Hit/Miss | |
|---|---|---|
| | | Block 0 |
| 0 | | |
| | | Block 1 |
| | | Block 2 |
| | | Block 3 |

# Example: Accessing A Direct-Mapped Cache

- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

| Mem Block | DM Hit/Miss |
|---|---|
| 0 | miss |
| 8 | miss |
| 0 | miss |
| 6 | miss |
| 8 | |



# Accessing a Direct-Mapped Cache

## Example

Show cache addressing for a byte-addressable memory with 32-bit addresses. Cache line $W$ = 16 B. Cache size $L$ = 4096 lines (64 KB).

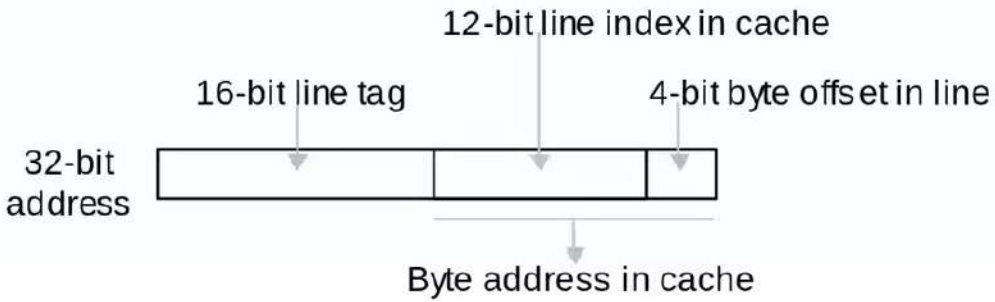
# Accessing a Direct-Mapped Cache

## Example

Show cache addressing for a byte-addressable memory with 32-bit addresses. Cache line $W$ = 16 B. Cache size $L$ = 4096 lines (64 KB).

**Solution**

Byte offset in line is $\log_2 16 = 4$ b. Cache line index is $\log_2 4096 = 12$ b. This leaves 32 – 12 – 4 = 16 b for the tag.




Components of the 32-bit address in an example direct-mapped cache with byte addressing.

# Direct Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = $2^{s+w}$ words or bytes
- Block size = line size – tag size = $2^{w}$ words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w} = 2^{s}$
- Number of lines in cache = m = $2^{r}$
- Size of tag = (s – r) bits

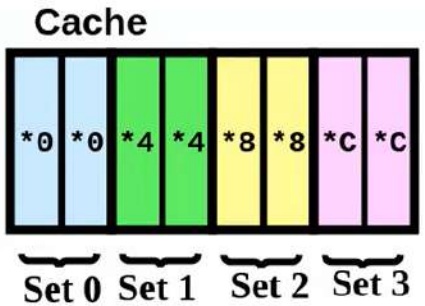
# Direct Mapping pros & cons

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

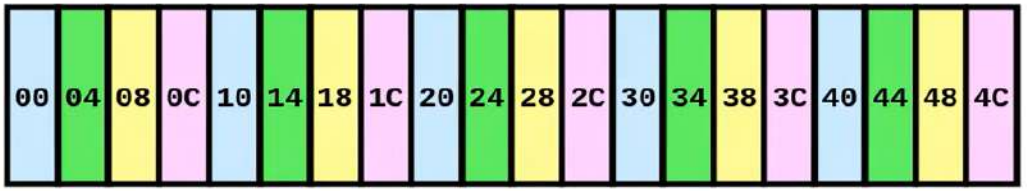
# Set Associative Mapping

- Cache is divided into a number of sets
- Each set contains k lines → k – way associative
- A given block maps to any line in a given set
  - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - 2 – way associative mapping
  - A given block can be in one of 2 lines in only one set

# Set-Associative Block Placement



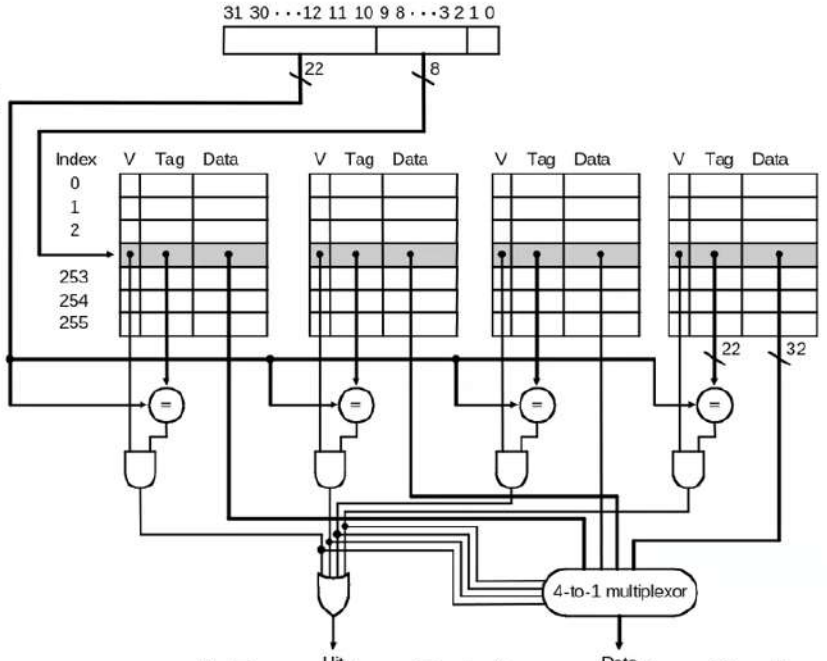

**address maps to set:**
location = *(block address* MOD # *sets in cache*)
(arbitrary location in set)




**Memory**

# Set Associative Cache Design

- Key idea:
  - Divide cache into sets
  - Allow block anywhere in a set
- Advantages:
  - Better hit rate
- Disadvantage:
  - More tag bits
  - More hardware
  - Higher access time




A Four-Way Set-Associative Cache

# Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 1: Mapping: 0 mod 2 = 0

| Mem Block | DM Hit/Miss | Set 0 | Mem[0] |
|---|---|---|---|
| 0 | miss | Set 1 | |

Set 0 is empty: write Mem[0] to Block 0

# Accessing a Set-Associative Cache

## Example

Show cache addressing scheme for a byte-addressable memory with 32-bit addresses. Cache line width $2^w$ = 16 B. Set size $2^s$ = 2 lines. Cache size $2^L$ = 4096 lines (64 KB).

# Accessing a Set-Associative Cache

## Example

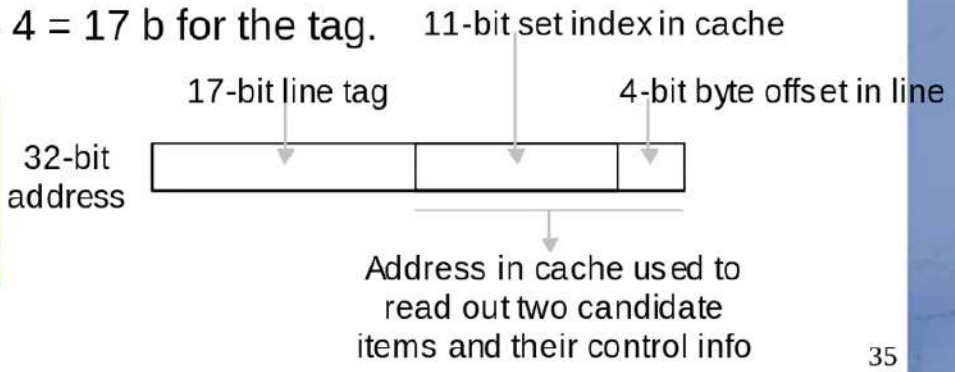
Show cache addressing scheme for a byte-addressable memory with 32-bit addresses. Cache line width $2^w$ = 16 B. Set size $2^s$ = 2 lines. Cache size $2^L$ = 4096 lines (64 KB).

**Solution**

Byte offset in line is $\log_2 16 = 4$ b. Cache set index is $(\log_2 4096/2) = 11$ b.

This leaves 32 – 11 – 4 = 17 b for the tag.

Components of the 32-bit address in an example two-way set-associative cache.



# Set Associative Mapping
## Address Structure

| Tag 9 bit | Set 13 bit | Word 2 bit |
|---|---|---|

- Use set field to determine which set of cache lines to look in (direct)
- Within this set, compare tag fields to see if we have a hit (associative)

e.g

| Address | Tag Data | | Set number |
|---|---|---|---|
| FFFFFC | 1FF | 12345678 | 1FFF |
| 00FFFF | 001 | 11223344 | 1FFF |

Same Set, different Tag, different Word

# e.g Breaking into Tag, Set, Word

- Given Tag=9 bits, Set=13 bits, Word=2 bits
- Given address $FFFFFD_{16}$
- What are values of Tag, Set, Word?
  - First 9 bits are Tag, next 13 are Set, next 2 are Word
  - Rewrite address in base 2: 1111 1111 1111 1111 1111 1101
  - Group each field in groups of 4 bits starting at right
  - Add *zero bits* as necessary to leftmost group of bits
- *0001* 1111 1111 *0001* 1111 1111 1111 *0001*
- → 1FF 1FFF 1 (Tag, Set, Word)