Direct-Mapped Cache: Write Allocate with Write-Through Protocol

WRITE data to address $[x]_{n-m}\,[w]_m[d]_b$

Block Address $A = [x]_{n-m}\,[w]_m$

Block size in bytes: $B = 2^b$

Cache size in blocks: $M = 2^m$ (2^{b+m} bytes)

Memory size in blocks $= 2^n$ (2^{b+n} bytes)

```
Compute cache index w = A mod M

 if (Cache Hit)
     1.   Write data  into byte d of cache[w].DATA
     2.   Store data into memory address [x]_{n-m} [w]_m[d]_b

if (Cache Miss)

     1.  Load block at memory block address A into cache[w].DATA
     2.  Update cache[w].TAG to x   ;cache[w].V = TRUE
     3.  Retry cache access
```

READ from address $[x]_{n-m}\,[w]_m[d]_b$

Cache Hit: Replace step 1 with Read word from the cache line and omit step 2

Direct-Mapped Cache: Write Allocate and Write Back

Write Allocate and Write-Back Protocol : write data to address $[x]_{n-m}$ $[w]_m$ $[d]_b$

Block Address A = $[x]_{n-m}$ $[w]_m$

```
Compute cache index w = A mod M
if  Cache Hit
          Write data  into byte  d of block cache[w].DATA
          Set cache[w].D to  TRUE
else  /* Cache Miss */
          Stall Processor
          if   cache block is dirty          /* cache[w].D = TRUE */
           Store cache[w].DATA into memory block at address [TAG][w]
          Load memory block at address [x][w]
          Update cache[w].TAG to x, cache[w].V = TRUE and cache[w].D to  FALSE
                    Retry cache Access
```

Direct-Mapped Cache: Reads in a Write Back Cache

Write-Back Protocol : read address $[x]_{n-m}$ $[w]_m$ $[d]_b$

- If cache hit read data field of cache entry
- If cache miss
    - replace current block writing it to memory if dirty
    - read in new block from memory and install in cache

```
Compute cache index w = A mod M
if  Cache Hit
            Read block cache[w].DATA; select word d of block
else   /* Cache Miss */
            Stall processor
            if   cache block is dirty   /* cache[w].D = TRUE  */
                        Store cache[w].DATA into memory at address [TAG][w]
            Read block at memory address A into cache[w].DATA
            Update cache[w].TAG to x, cache[w].V to TRUE, cache[w].D to FALSE
            Retry cache access
```

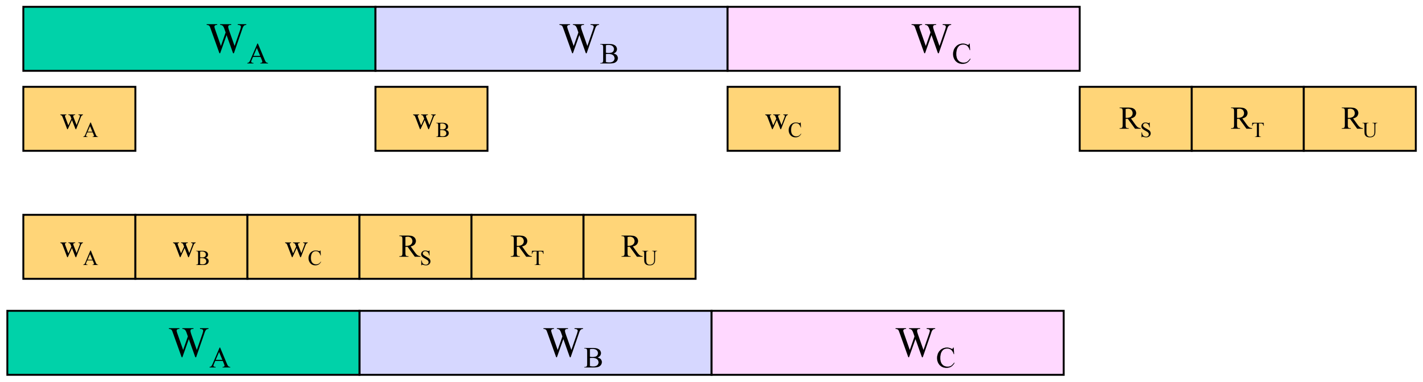
Direct-Mapped Cache: Write Allocate with Write-Through

Write Allocate and Write-Through Protocol**:** write data to address $[x]_{n-m}\ [w]_m[d]_b$

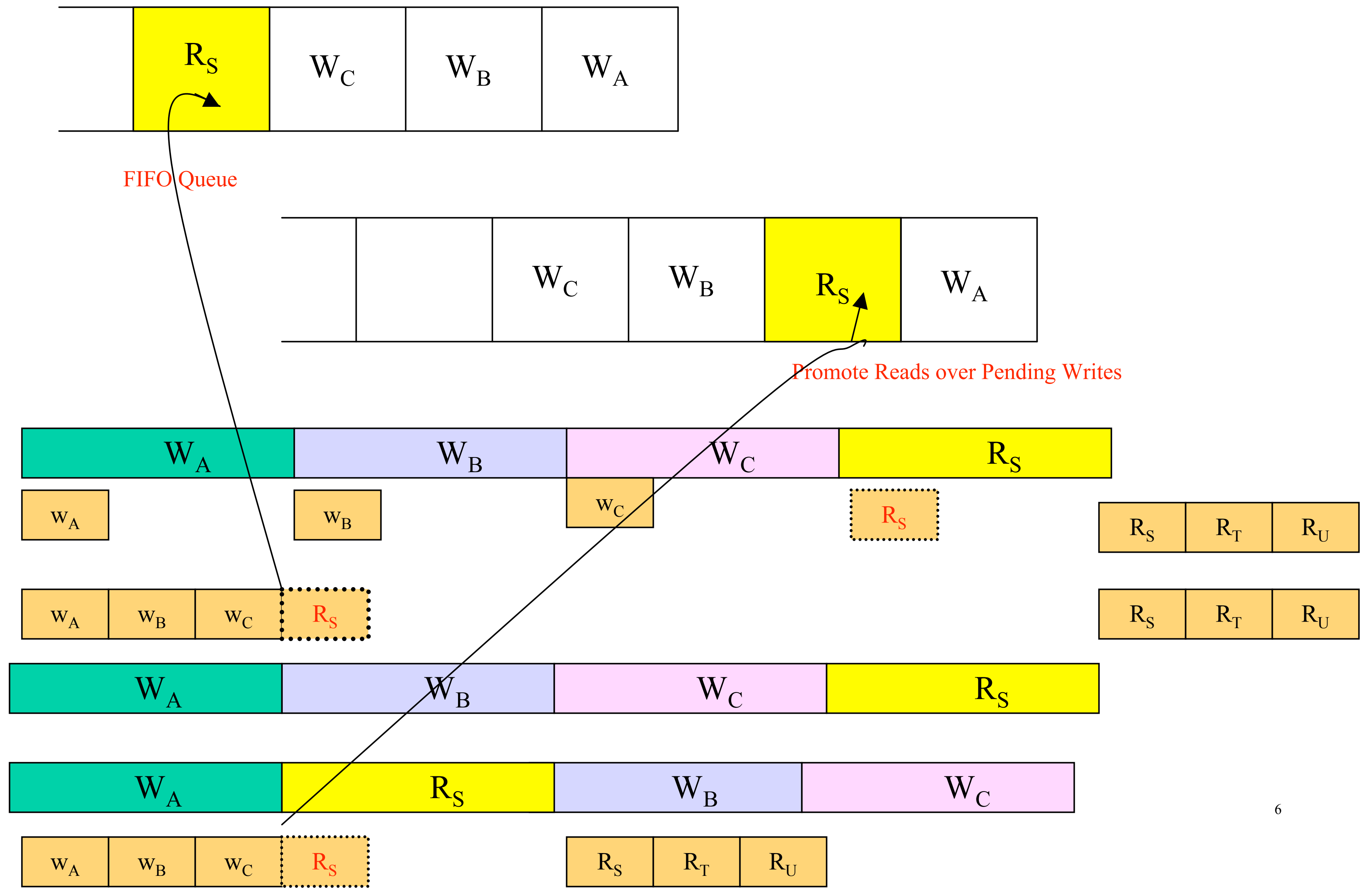
Block Address $A = [x]_{n-m}\ [w]_m$

- Synchronous Writes
- Writes proceed at the speed of main memory not at speed of cache

$W_A \quad W_B \quad W_C \quad R_S \quad R_T \quad R_U$



Direct-Mapped Cache: Write Allocate with Write-Through



Direct-Mapped Cache: Write Allocate with Write-Through

Write Allocate and Write-Through Protocol**:** write data to address $[x]_{n-m}\ [w]_m[d]_b$

Block Address $A = [x]_{n-m}\ [w]_m$

- Writes proceed at the speed of main memory not at speed of cache
- To speed up writes use asynchronous writes:
    - Write into cache and simultaneously into a write buffer
    - Execution continues concurrently with memory write from buffer
    - Write buffer should be deep enough to buffer burst of writes
    - If write buffer full on write then stall processor till buffer frees up
    - Write buffer served in FCFS order : simple protocol
    - Allow (later) reads to overtake pending writes
        - Read protocol modified appropriately
        - On memory read check write buffer for a write in transit

Writes Summary

1. In a write allocate scheme with a write through policy:
   - Write Hit: Update both cache and main memory (1W)
   - Write Miss: Read in block to cache. Update cache and main memory (1R + 1W)

2. In a write allocate scheme with a write back policy:
   - Write Hit: Update cache only
   - Write Miss: Read in block to cache. Write evicted block if dirty. Update cache. (1R + 1W if dirty block being replaced)

3. In a no write allocate scheme with a write through policy:
   - Write Hit: Update both cache and main memory (1W)
   - Write Miss: Update main memory only (1W)

4. In a no write allocate scheme with a write back policy:
   - Write Hit: Update cache only
   - Write Miss: Update main memory only (1W)

Set-Associative Organization

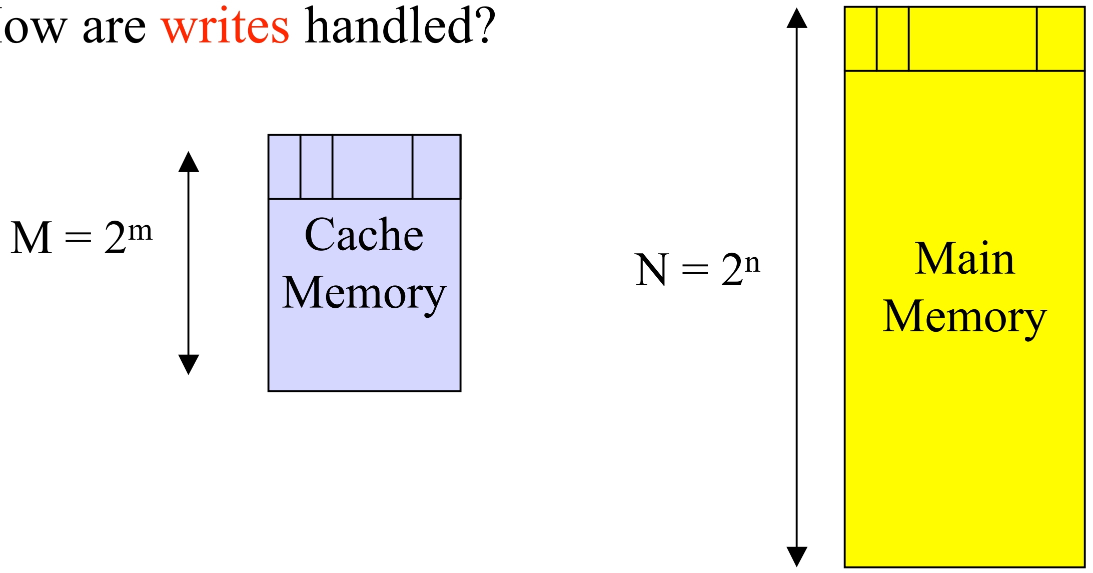
Cache Organization: Main memory address: $n+b$ bits

2^m cache blocks vs 2^n blocks of main memory, $n > m$

Block consists of 2^b consecutive bytes

Four Basic Questions:

1. Where in cache do we place a block of main memory?
2. How do we locate (search) for a memory reference in the cache?
3. Which block in the cache do we replace?
4. How are writes handled?



Set-Associative Cache: Motivation

Direct Mapped Cache:

1. Only one cache location to store any memory block

Conflict Misses: cache forces eviction even if other cache blocks unused

Improve miss ratio by providing choice of locations for each memory block

Fully Associative Cache:

1. Any cache location to store any memory block

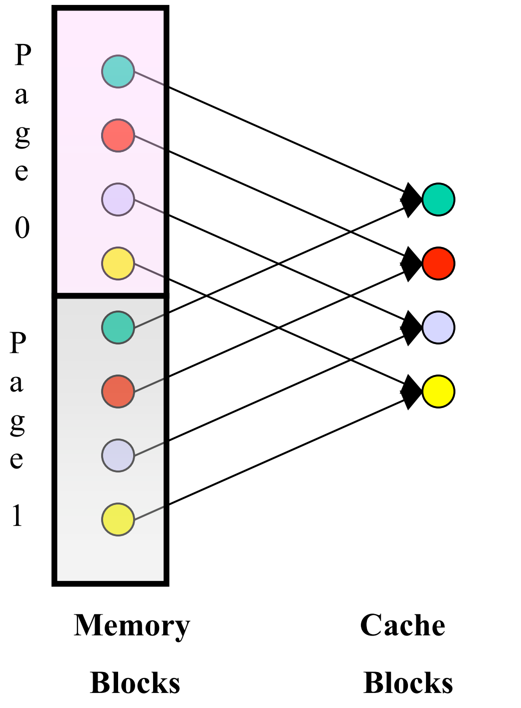
Reduce Conflict Misses improving Miss ratio

No Conflict Misses in a Fully Associative Cache

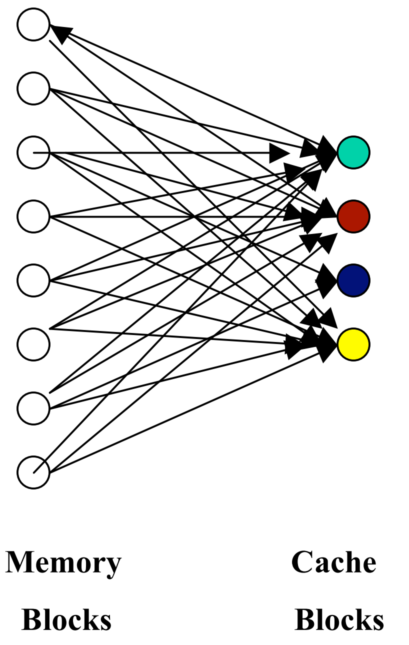
Set Associative Cache

Compromise between miss rate and complexity (power, speed)

Direct Mapped and Fully Associative Cache Organizations



Direct-Mapped Cache mapping



Fully Associative mapping

- All cache blocks have different colors
- Memory blocks in each page cycle through the same colors in order
- A memory block can be placed only in a cache block of matching color

- A memory block can be placed in any cache block

Set-Associative Cache: Motivation

Direct Mapped Cache:

- Only one cache location to store any memory block
- Single collision: cache forces eviction even if other cache blocks unused
- Improve miss ratio by providing choice of locations for each memory block

Example: Cache size = M words

Therefore memory words with addresses M apart will map to the same cache block in a DM cache

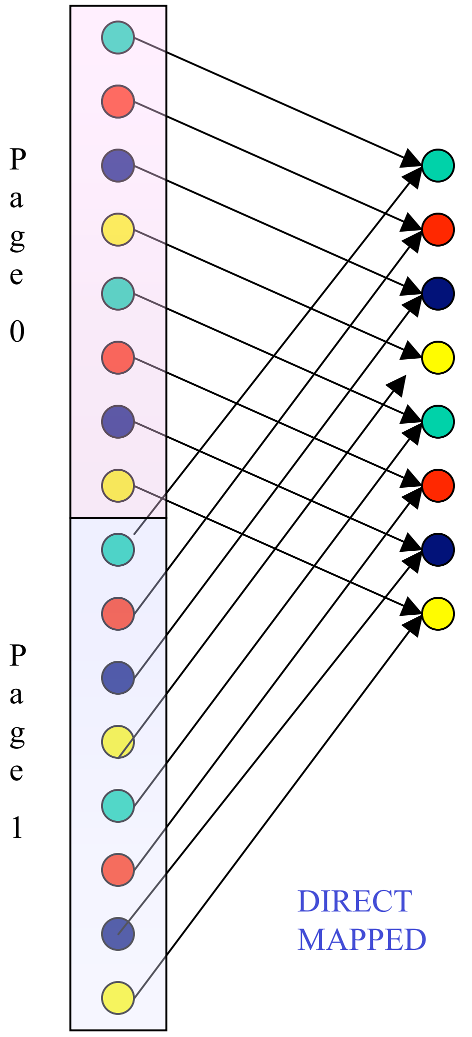
```
while (!done) {
for (i = M; i < limit; i = i+M)
 a[i] += (a[i-M] + a[i+M]) / 2;
}
```

a[i] += (a[i-M] + a[i+M]) all map to same cache index: (i mod M)

Every memory access in every iteration could be a cache miss

Reduce Conflict Misses using set associative cache

Mapping between Memory Blocks and Cache Blocks

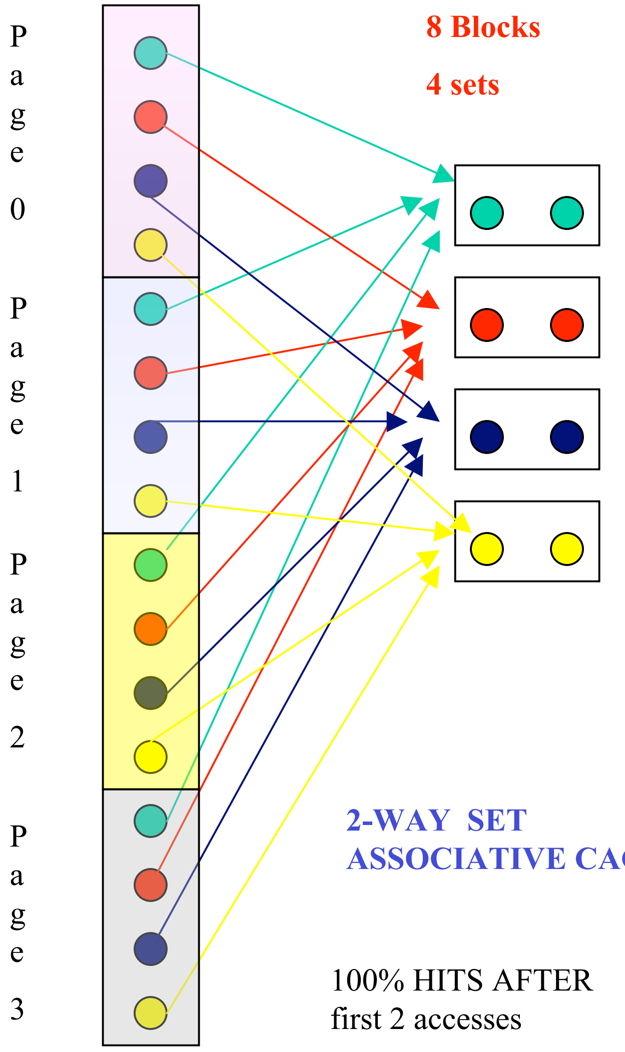


Cache Size:
8 Blocks

DIRECT MAPPED

EXAMPLE: 0,8,0,8,0,8,……

100% MISS



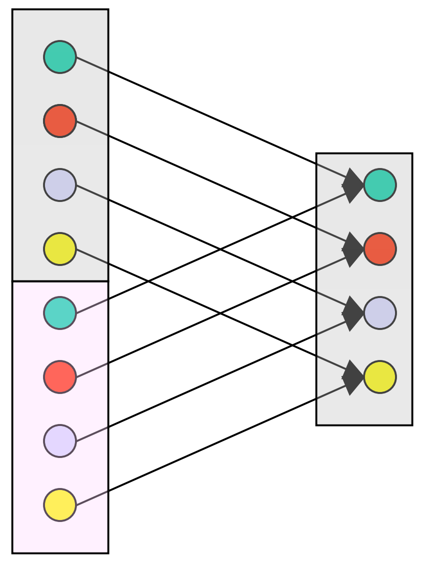
Cache Size:
8 Blocks

4 sets

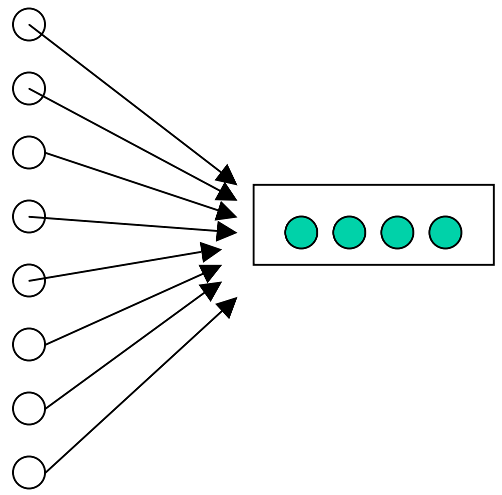
2-WAY SET ASSOCIATIVE CACHE

100% HITS AFTER first 2 accesses

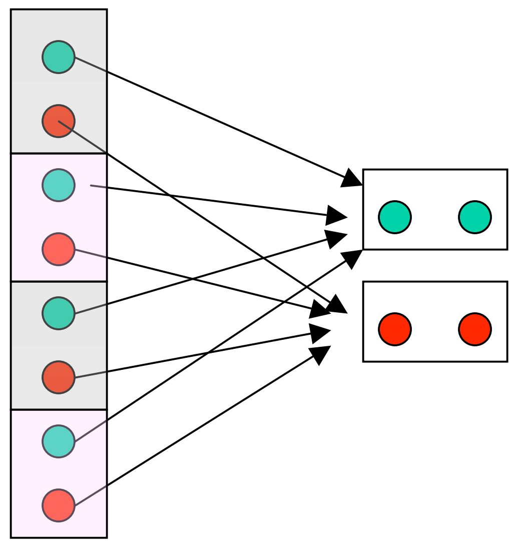
Mapping between Memory Blocks and Cache Blocks



Direct-Mapped Cache mapping



Fully Associative mapping



2-way Set Associative mapping

- Cache blocks grouped in sets
- Page size equals number of sets
- All sets of the cache have different colors
- All blocks within a set have the same color
- Number of blocks in set defines "way" of the cache
- A memory block can be placed only in set of matching color

- All cache blocks have different colors
- Memory blocks in any page cycle through the same colors in order
- A memory block can be placed only in a cache block of matching color

Set-Associative Cache

K-way Set Associative Cache:

Cache size: $M = 2^m$ blocks

Cache divided into sets of size $K = 2^k$ blocks each (K-way set associative)

Cache consists of $S = 2^s = 2^{m-k}$ sets

Page Size = S blocks

A block in a page is mapped to exactly one set

Memory block with address A mapped to the unique set: (A mod S)

Memory block may be stored in any cache block in the set

With each cache block store a tag of (n - s) MSBs of memory address A

Example:

Cache size: M = 32 blocks,

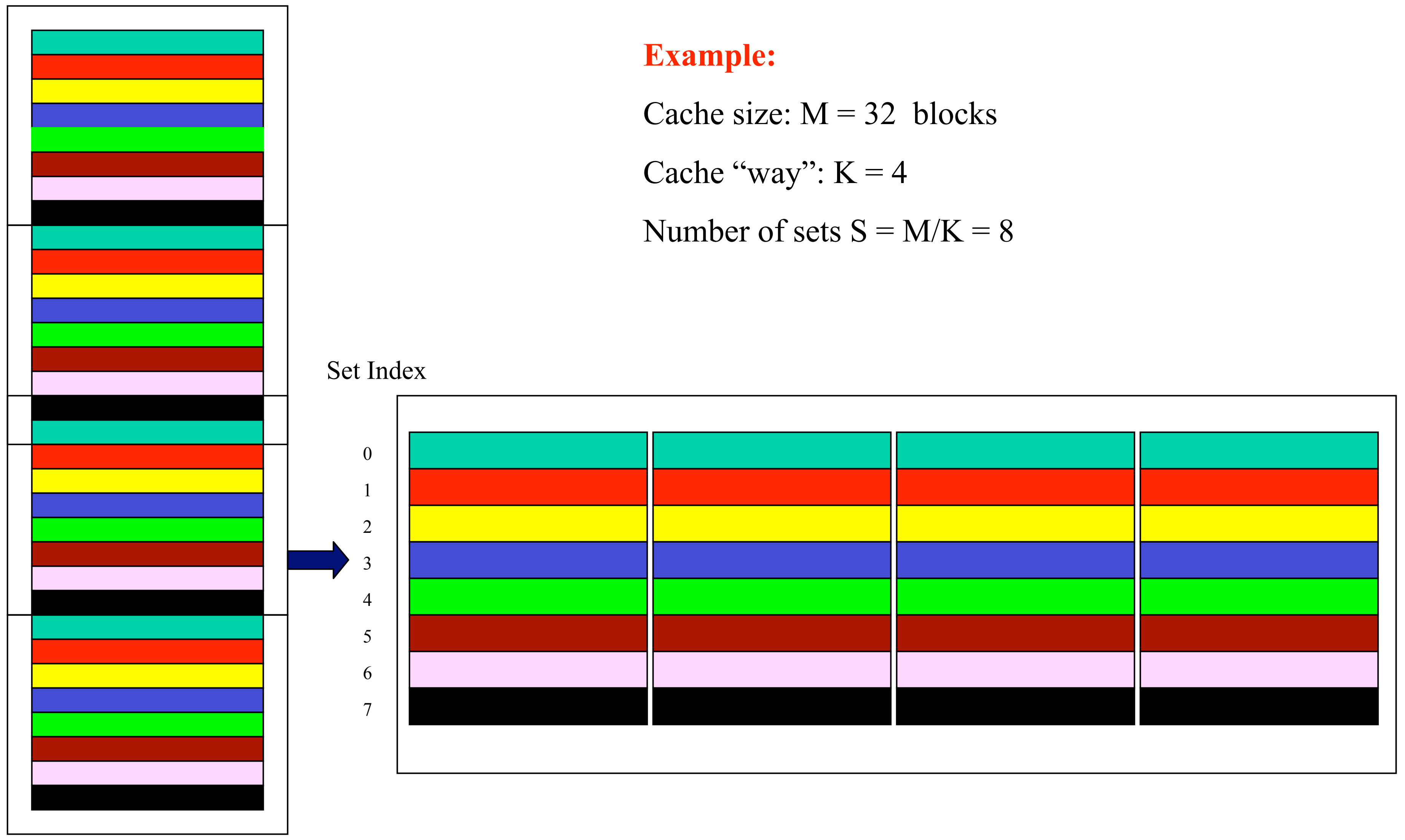
Cache "way": K = 4

Number of sets: S = M/K = 8

Consider address trace 0, 32, 64, 96, 128, …….

In Direct mapped cache (K=1) all blocks mapped to cache block 0

In this example (K=4) all blocks mapped to set 0; but 4 cache blocks available in each set



Example:

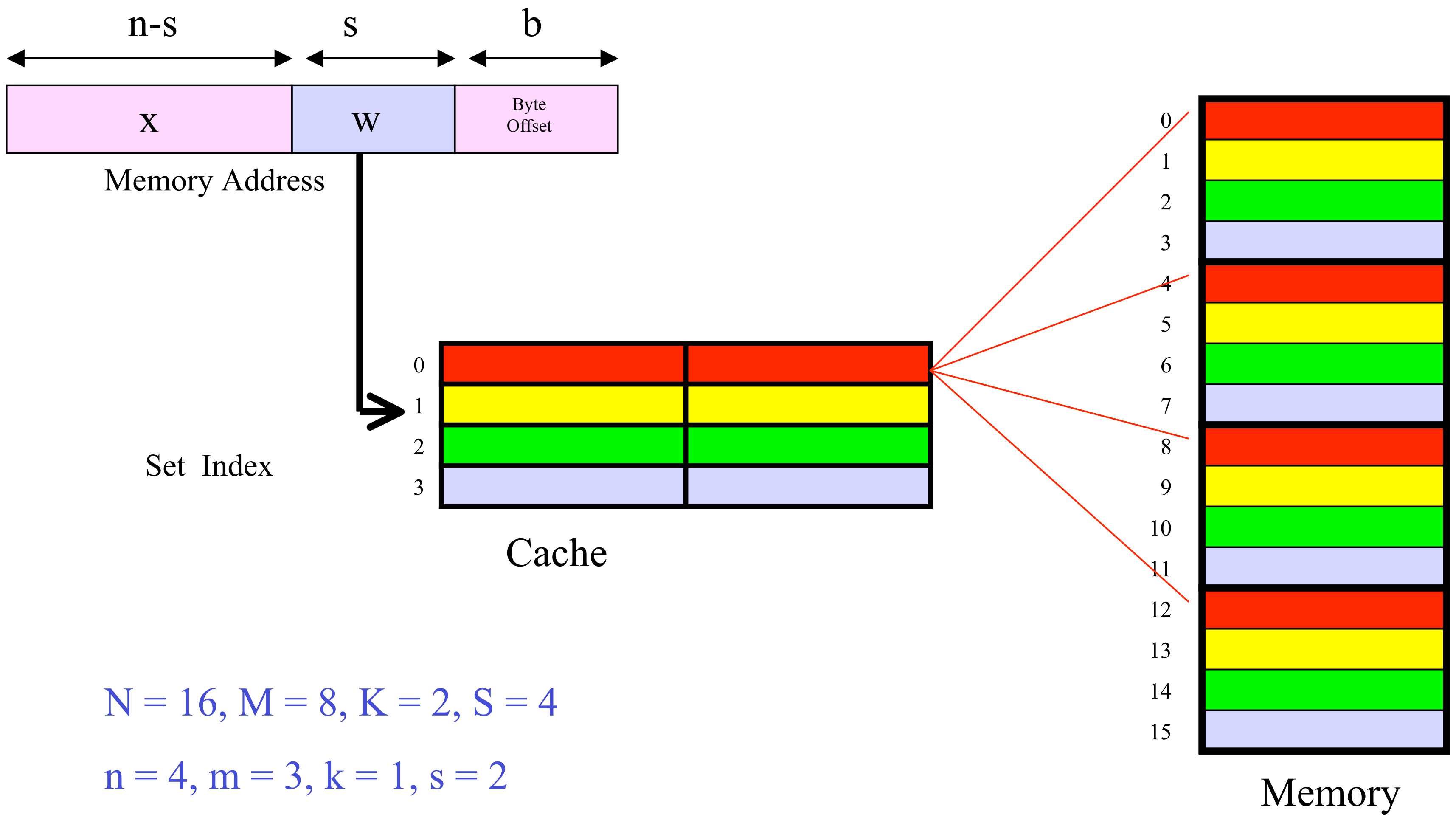
Cache size: M = 32 blocks

Cache “way”: K = 4

Number of sets S = M/K = 8

Cache

K-way Set-Associative Cache (K = 2)



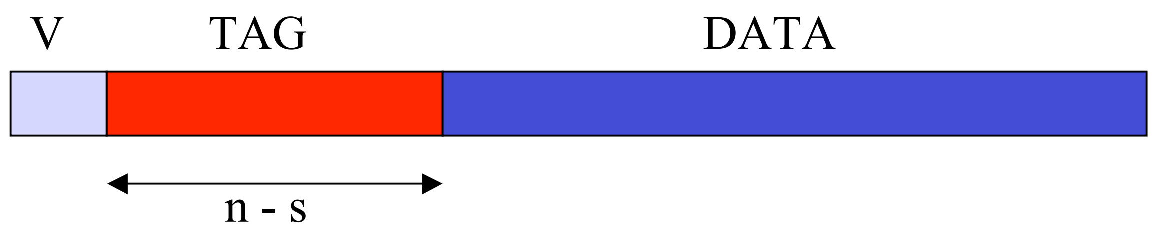
N = 16, M = 8, K = 2, S = 4

n = 4, m = 3, k = 1, s = 2

Set-Associative Cache Organization

To identify which of the 2^{n-s} possible memory blocks is actually stored in a given cache block, each cache block is given a TAG of *n-s* bits.

Cache Entry:

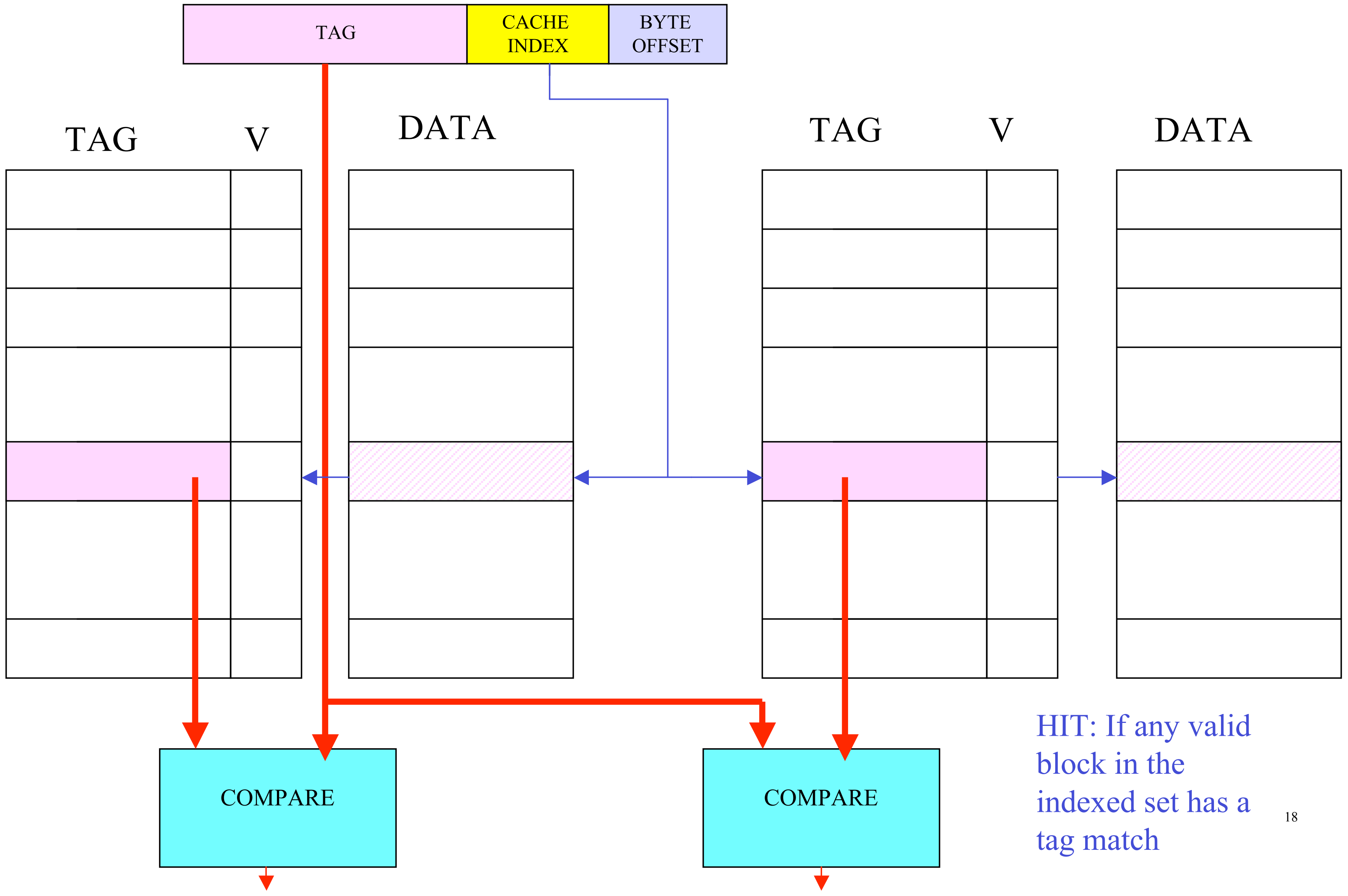


V (Valid) bit: Indicates that the cache entry contains valid data

TAG : identifies which of the 2^{n-s} memory blocks stored in cache block

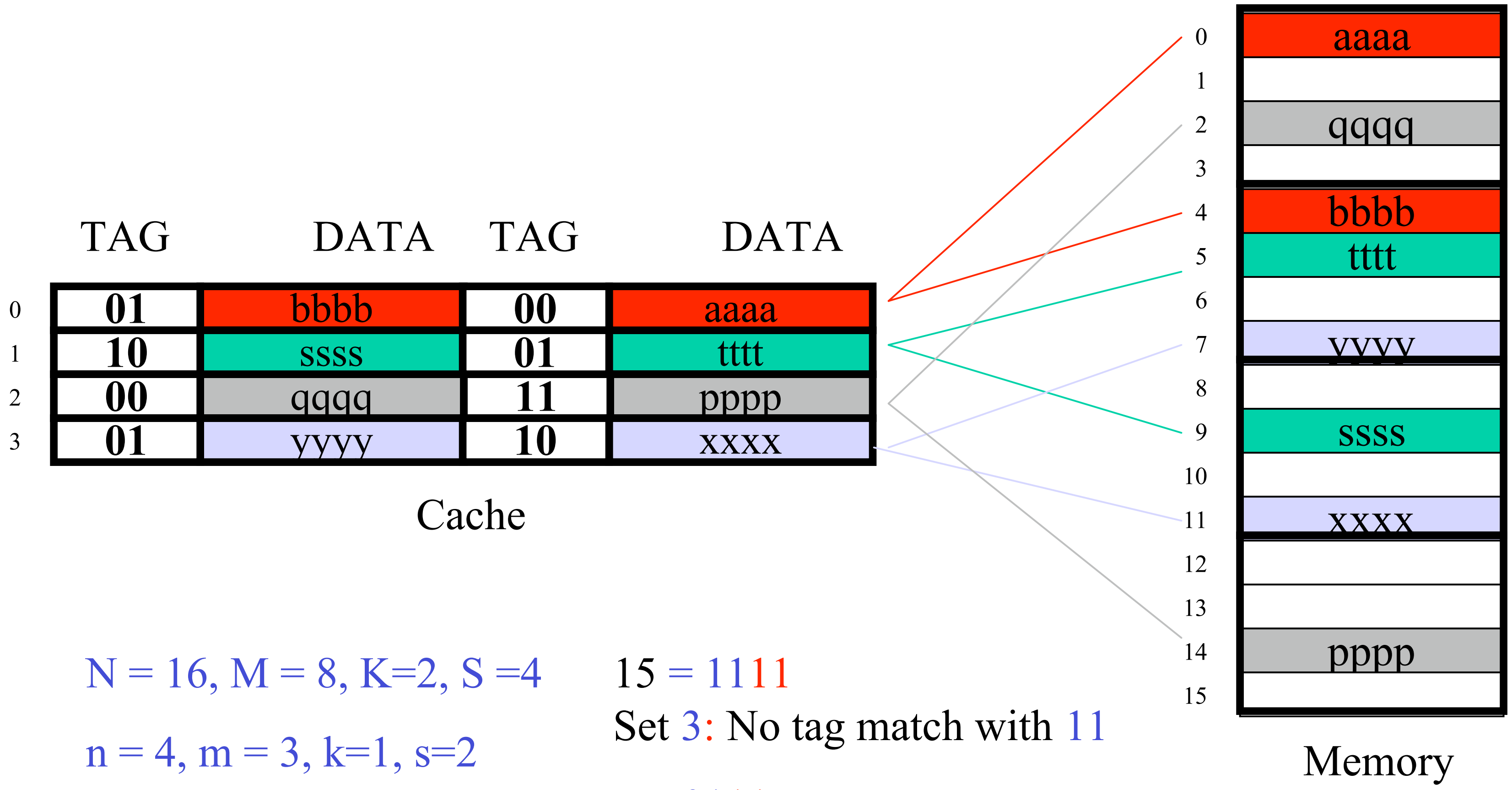
DATA : Copy of the memory block stored in this cache block

2-way Set Associative Cache



HIT: If any valid block in the indexed set has a tag match

Set-Associative Cache Organization



N = 16, M = 8, K=2, S =4

n = 4, m = 3, k=1, s=2

15 = 1111

Set 3: No tag match with 11

7 = 0111

Set 3: Tag match with 01

Set-Associative Cache: Operation

Assume write through (so all blocks are clean)

Memory Read Protocol: n-bit memory block address A = $[x]_{n-s}$ $[w]_s$

```
Compute cache set index w = A mod S
Read all K blocks in set cache[w]
Simultaneously check tags against x
 if cache hit
        Read DATA field of matching block into processor
else /* cache miss : no block in set matches */
        Stall processor till block brought into cache
        Choose a victim block in set cache[w] to evict from the cache
        Load main memory block at address A into DATA field of victim
        Update TAG field of cache block to x and V to TRUE
        Restart processor from start of cycle
```

Cache Hit if there is a block in set cache[w] such that its V bit is set and its TAG field matches x

Require K comparators to compare tags simultaneously

Set-Associative Cache: Example

	TAG	DATA	TAG	DATA
0	**00**	AAAA		
1				
2				
3				

	TAG	DATA	TAG	DATA
0	00	AAAA		
1				
2	**01**	BBBB		
3				

	TAG	DATA	TAG	DATA
0	00	AAAA	**01**	CCCC
1				
2	01	BBBB		
3				

Cache

Address Trace: 0, 6, 4,0, 8

0000: Set 00 Tag: 00 AAAA

0110: Set 10 Tag: 01 BBBB

0100: Set 00 Tag: 01 CCCC

0000: Set 00 Tag: 00 AAAA

Hit!

1000: Set 00 Tag: 10 DDDD

Replacement needed!

Set-Associative Cache Replacement

Replacement Strategy:

Which of the K blocks in the selected set is replaced?

Random: One of the K blocks in the set chosen at random and replaced

LRU (Least Recently Used) Policy: Replace the block that has not been referenced for the longest time -- block whose last reference most in the past

Difficult to implement efficiently in hardware

Approximations to LRU often used

In example: 0 referenced more lately than 4: replace 4

	TAG	DATA	TAG	DATA
0	00	AAAA	10	**DDDD**
1				
2	01			
3				

Cache

Set-Associative Cache: Write Allocate with Write-Through

Write Allocate and Write-Through Protocol: write data to address $A = [x]_{n-s} [w]_s$

```
Compute cache *set* index w = A mod S
*Search for match among blocks in set cache*[w]

if cache hit
        Write data into DATA field of matching block
        Store data into memory address A
 else /* cache miss */
        Stall processor
        Select victim to replace from set cache[w]
        Load cache entry of victim with memory block at A
        Update fields TAG to x and V to TRUE
        Restart cache access
```

Set-Associative Cache: Write Allocate with Write Back

Write Allocate and Write-**Back Protocol** : **write** data to address $A = [x]_{n-s}\,[w]_s$

If cache hit update data field of cache block

If cache miss

select a block to replace writing it to main memory if dirty

update cache block with new data and V, D, TAG fields

```
Compute cache set index  w = A mod S
if  cache hit

Write data  into DATA field of  matching block
            Update D field to TRUE
else  /* cache miss */
            Stall processor
            Choose a victim block in set cache[w] to replace from the cache
            if   victim block is dirty
                  Store DATA field of victim into memory at address [tag][w]
                  Load memory block at A into victim entry of cache
                  Update TAG to x,  V = TRUE ,  D fields to FALSE
            Restart cache access
```

Set-Associative Cache: Reads in a Write Back Cache

Write-Back Protocol : **read** address $A = [x]_{n-s}\,[w]_s$

If cache hit read data field of cache block

If cache miss

select a block to replace writing it to memory if dirty

read in new block from memory and install in cache

```
Compute cache index set  w = A mod S
if cache hit
        Read cache[w].DATA into processor
else  /* cache miss */
        Stall processor
        Choose a victim block in set cache[w] to replace from the cache
        if  victim block is dirty
            Store DATA field of victim into memory at address [tag][w]
        Load block at memory address A into DATA  field of selected block
        Update fields of selected block: TAG to x, V to TRUE, D to FALSE
        Restart processor
```