Lecture 7

- Cache Memory Principles
- Error Correction

Cache Memory Principles

A cache memory is a fast, small memory. It is placed between the CPU and main memory as shown in Figure 18.

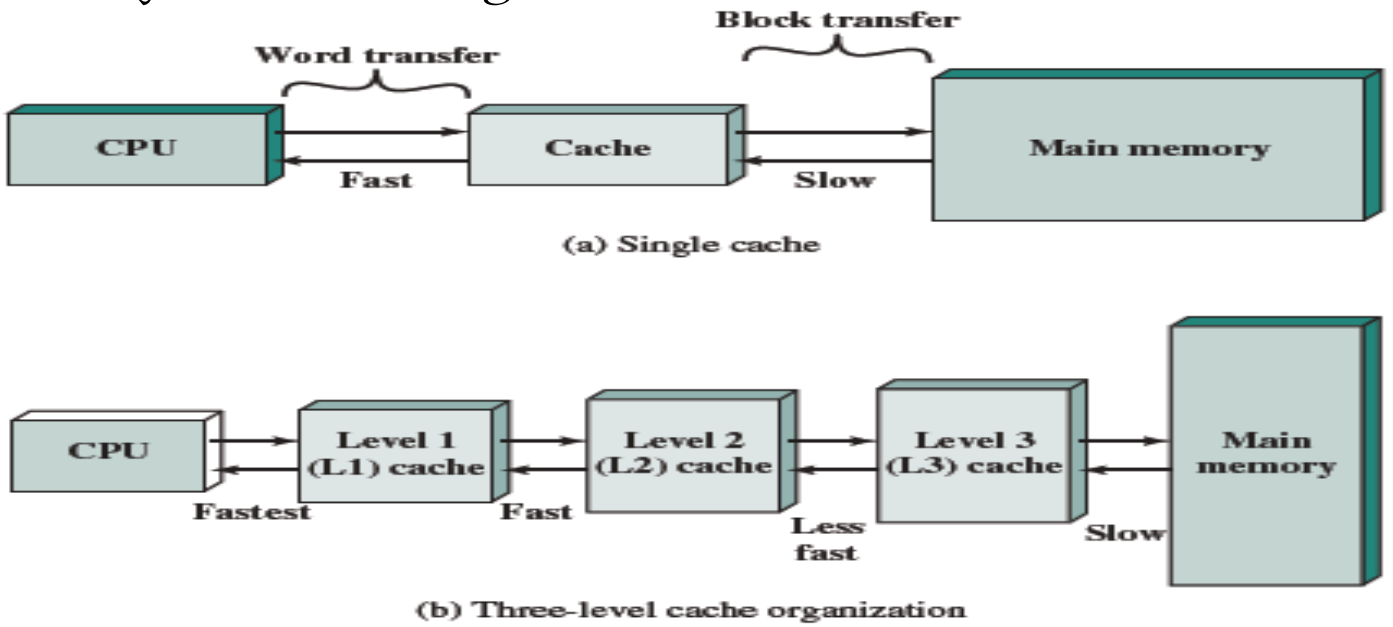


Figure 18: Cache and Main Memory

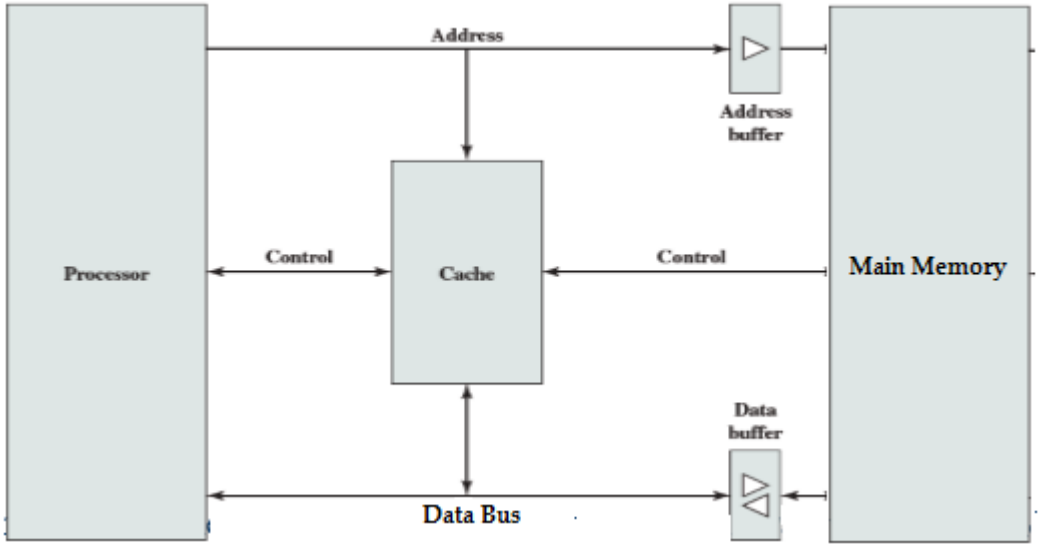
The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The differences between the cache memory and the main memory are as follows:

Cache memory	Main memory
Non Sequential Addresses	Sequential address
Non fixed location for addresses	A fixed location for each address

Cache Memory Principles :

- The cache memory is also called content addressable because the cache memory stores both address and content (data).
- Figure 3 depicts the typical cache organization:



- The performance measure of cache memory called **hit ratio**. When the CPU refers to memory and finds the word in cache, it is **said to produce a hit**. If the word is not found in cache, then it is in main memory and it counts as a **miss**.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache.

The main memory is accessed to read the word. Figure 21 illustrates the read operation.

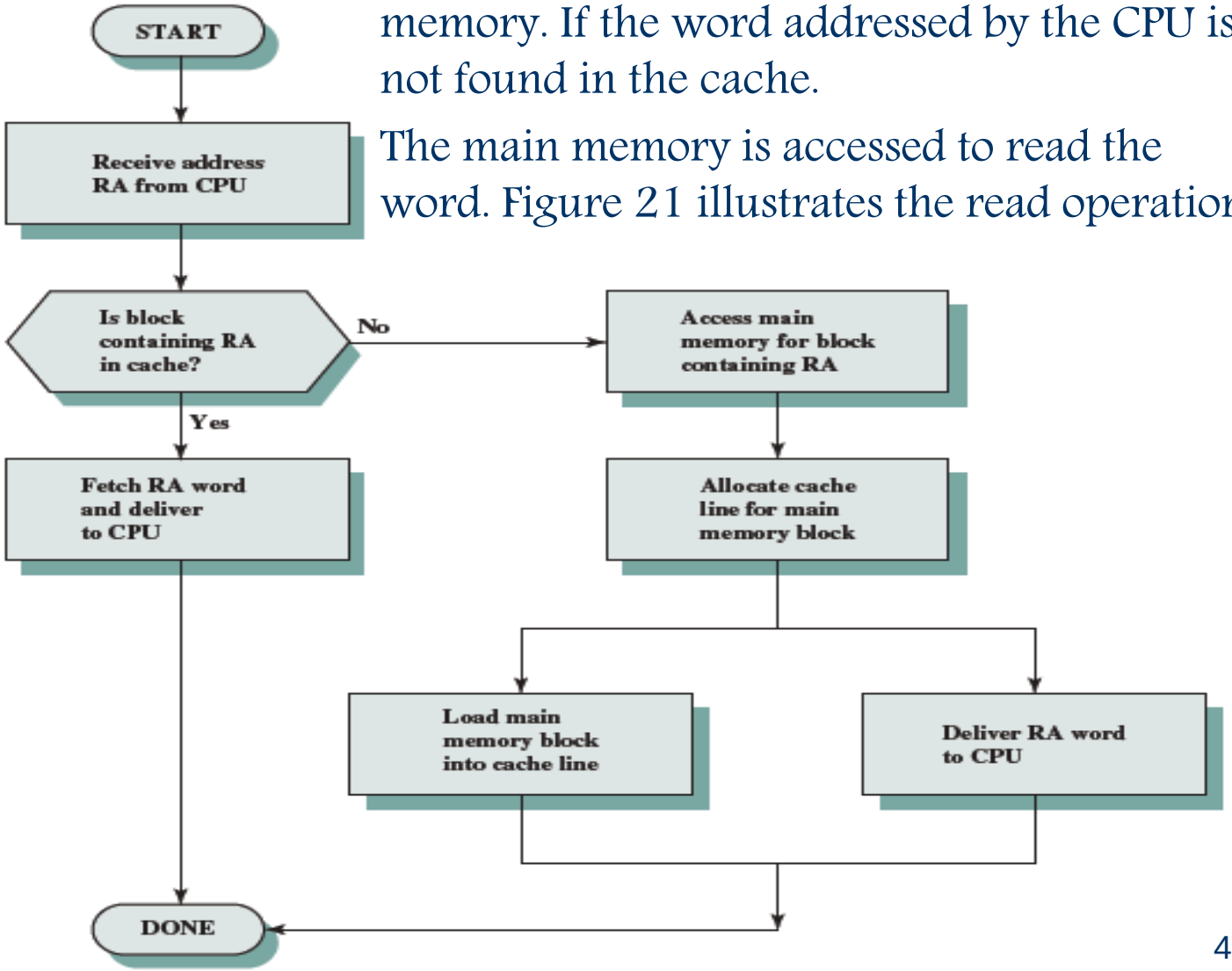


Figure 21 Cache Read Operation

Mapping in Cache Memory:

- The transformation of data from main memory to cache memory is referred to as a **mapping** process. Three types of mapping procedures are of practical interest in considering the organization of cache memory:

1) Direct mapping.

2) Associative mapping.

3) Set-associative mapping.

2) Direct mapping.

The CPU address is divided into **two fields, index and tag** fields. The number of bits in the index field is **equal to the number of address bits required to access the cache memory**.

Address		Content
Index	Tag	
00	00	C
	01	A
	10	B
	11	D
01	.	.
	.	.
	.	.

The disadvantage of direct mapping is that the hit ratio may drop considerably if two or more words with addresses having the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range.

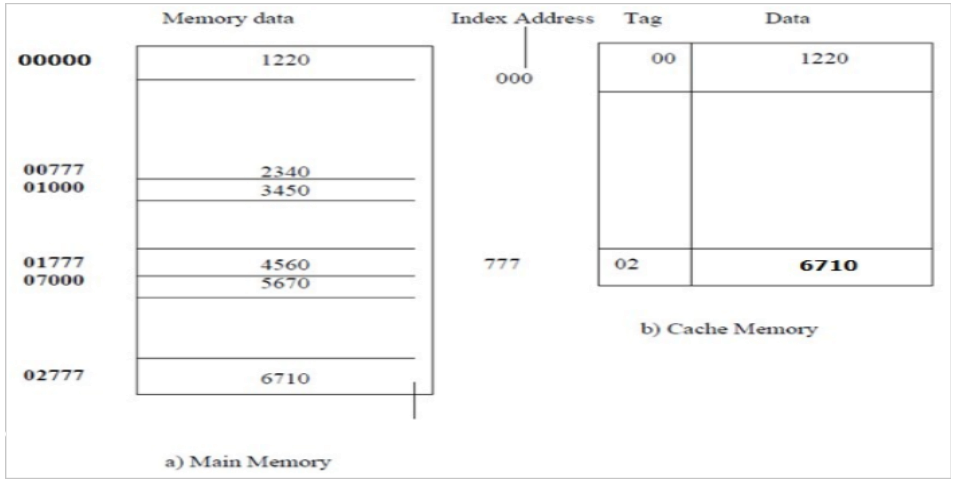


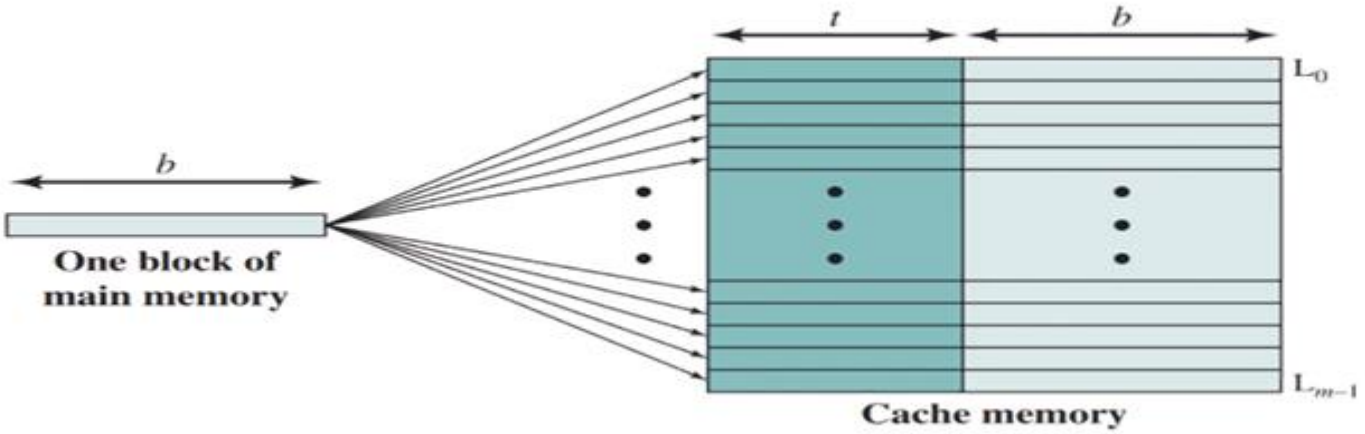
Figure 4.8 Mapping from Main Memory to Cache:

- 1) Associative mapping:

– The associative memory stores both the address and content (data) of the memory.

– The associative memory is searched for a matching address. If the address is found, the corresponding data is read and sent to the CPU.

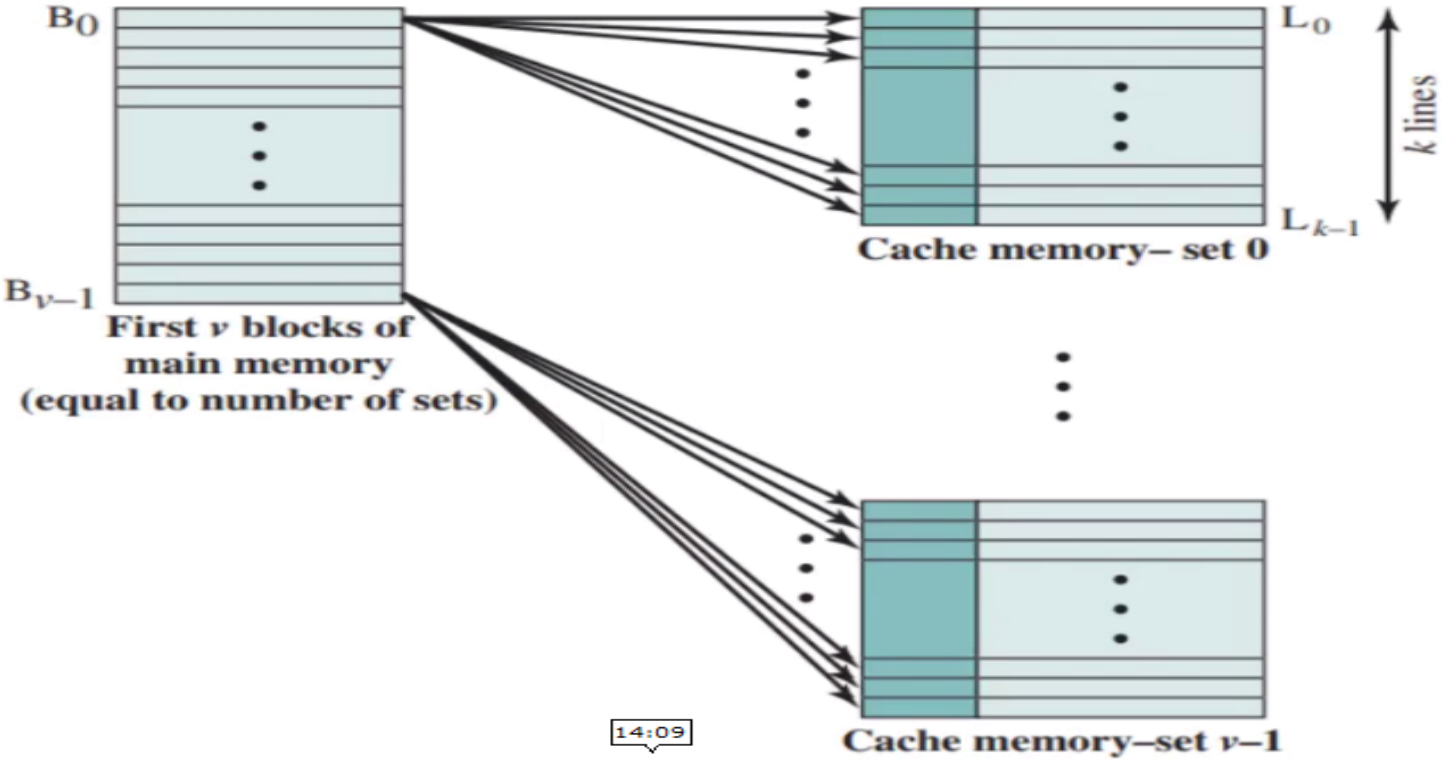
– If the cache is full, a simple procedure is to replace the old word in cache by a new word. This constitutes a first-in-first-out (FIFO) replacement policy.



- **3) Set-associative mapping**

  In set-associative mapping, each word of cache can store two or more words of memory under the same index address.

- Each data word is stored together with its tag, and the number of tag-data items in one word of cache is said to form a set. When the CPU address is then compared with both tags in the cache to determine if a match occurs. The hit ratio will improve as the set size increases.



Write methods in Cache Memory:

The update of data in cache memory is referred to as write methods. Two types of write methods are:

1) Write-through method.
2) Write-back method.

1) Write-through method:

− In which first update content in the main memory and second update content in the cache memory if its address already in cache.

2) Write-back method:

− In which first updates content in the cache memory if its address is found in cache and second updates content in the main memory if its address is not found in cache.

Direct Mapping Calculation:

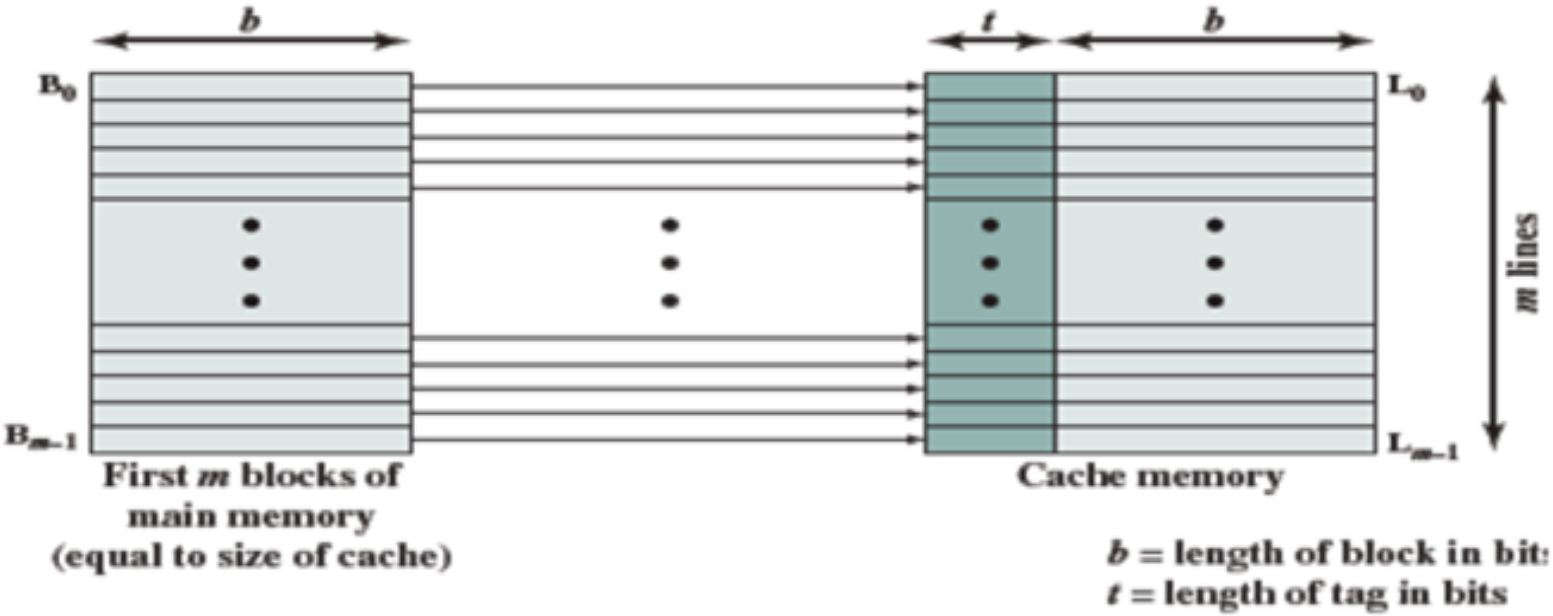
The simplest technique, known as direct mapping, maps **each block of main memory into only one possible cache line**. The mapping is expressed as:

$$i = j \text{ modulo } m$$

where

i = cache line number
j = main memory block number
m = number of lines in the cache

Figure 4.8a shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks



(a) Direct mapping

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

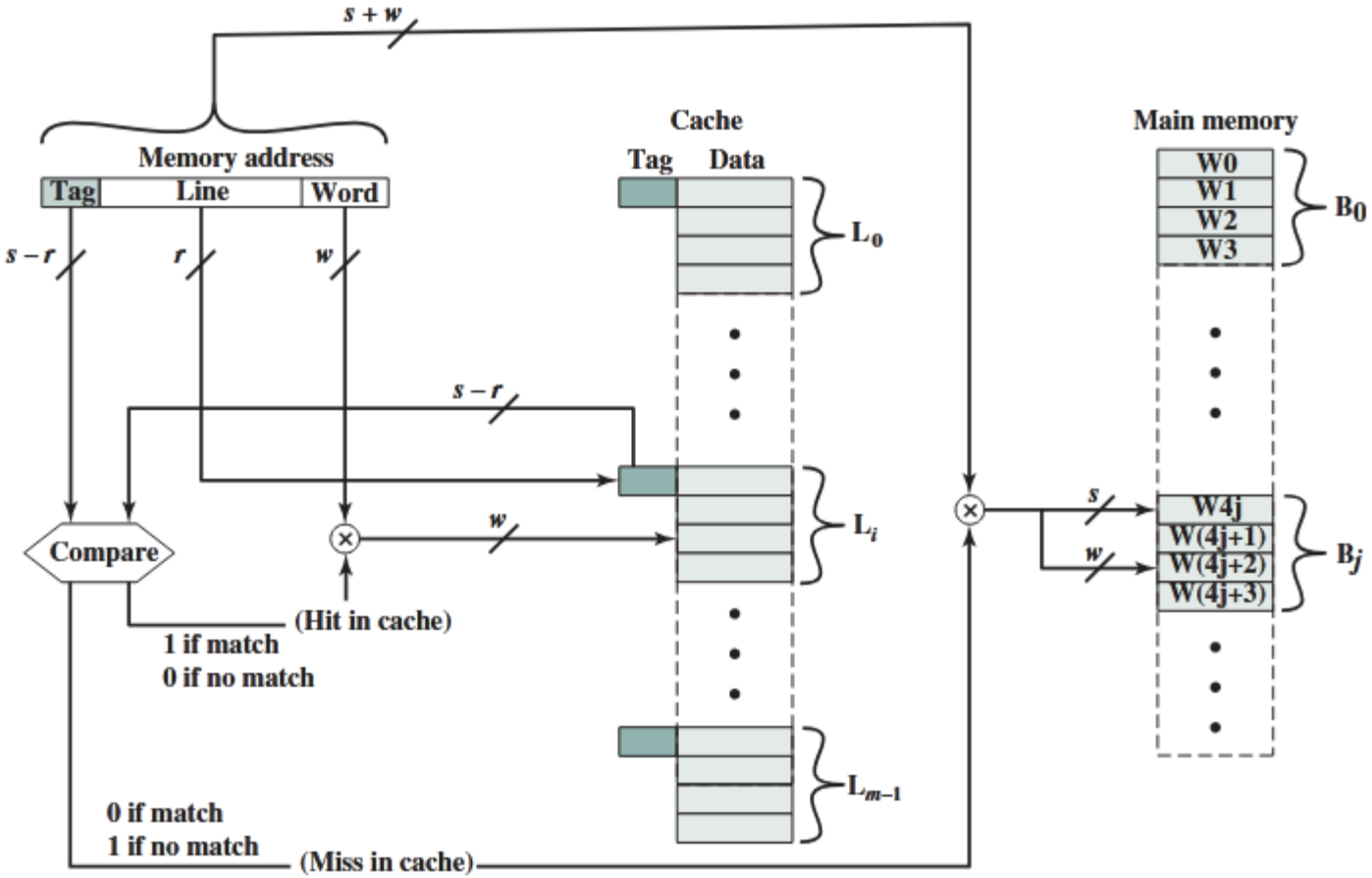


Figure 4.9 Direct-Mapping Cache Organization

Address divided into three parts:

Tag	Line	Word

- Memory address size = (s+w)
- Word part = w
- Line part = r
- Tag part = s-r

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m+1, 2m+1, \ldots, 2^s - m + 1$
⋮	⋮
$m-1$	$m-1, 2m-1, 3m-1, \ldots, 2^s - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant s - r bits serve this purpose.

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache.

Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 4.8b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match. Figure 4.11 illustrates the logic

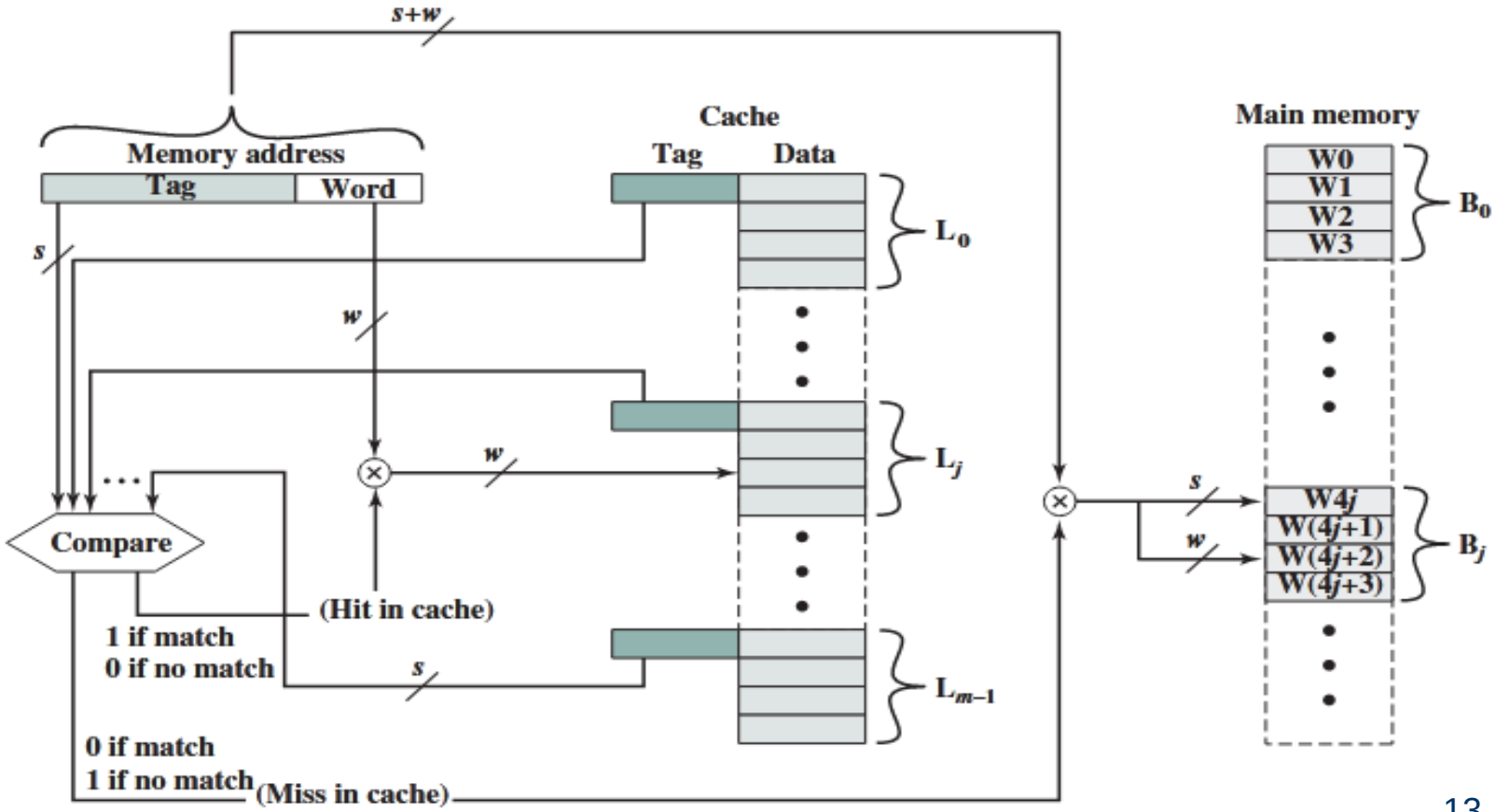


Figure 4.11 Fully Associative Cache Organization

Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. To summarize,

- Address length = (s + w) bits
- Number of addressable units = 2s+w words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = 2s + w

  2w = 2s
- Number of lines in cache = undetermined
- Size of tag = s bits

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel>

Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

In this case, the cache consists of number sets, each of which consists of a number of lines. The relationships are

$$m = v * k$$

$$i = j \text{ modulo } v$$

where

i = cache set number

j = main memory block number

m = number of lines in the cache

v = number of sets

k = number of lines in each set

Memory address size = (s+w)

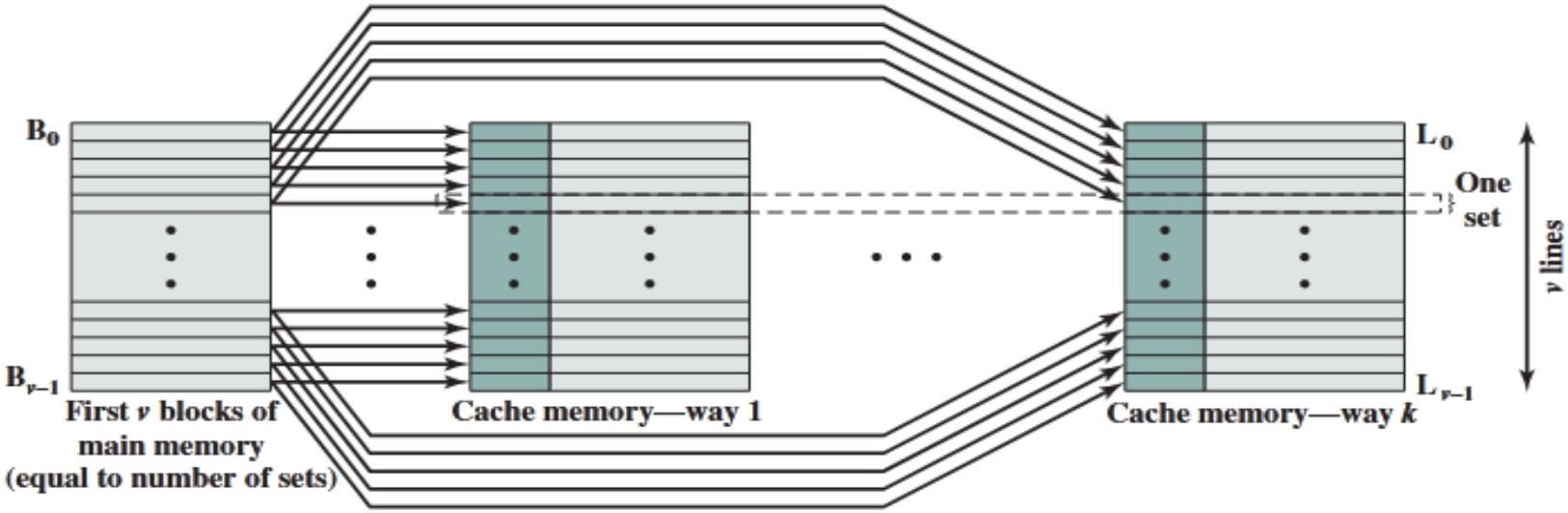
Word part = w

Line part = d

This is referred to as k-way set-associative mapping. block B j can be mapped into any of the lines of set j. Figure 4.13a illustrates this mapping for the first v blocks of main memory.

As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B 0 maps into set 0, and so on.

Thus, the set-associative cache can be physically implemented as v associative caches. It is also possible to implement the set-associative cache as k direct mapping caches, as shown in Figure 4.13b. Each direct-mapped cache is referred to as a way, consisting of v lines. The first v lines of main memory are direct mapped into the v lines of each way; the next group of v lines of main memory are similarly mapped, and so on.



(b) k direct–mapped caches

Figure 4.13 Mapping from Main Memory to Cache: *k*-Way Set Associative

- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set = k
- Number of sets = $v = 2^d$
- Number of lines in cache = $m = kv = k \times 2^d$
- Size of cache = $k \times 2^{d+w}$ words or bytes
- Size of tag = $(s - d)$ bits

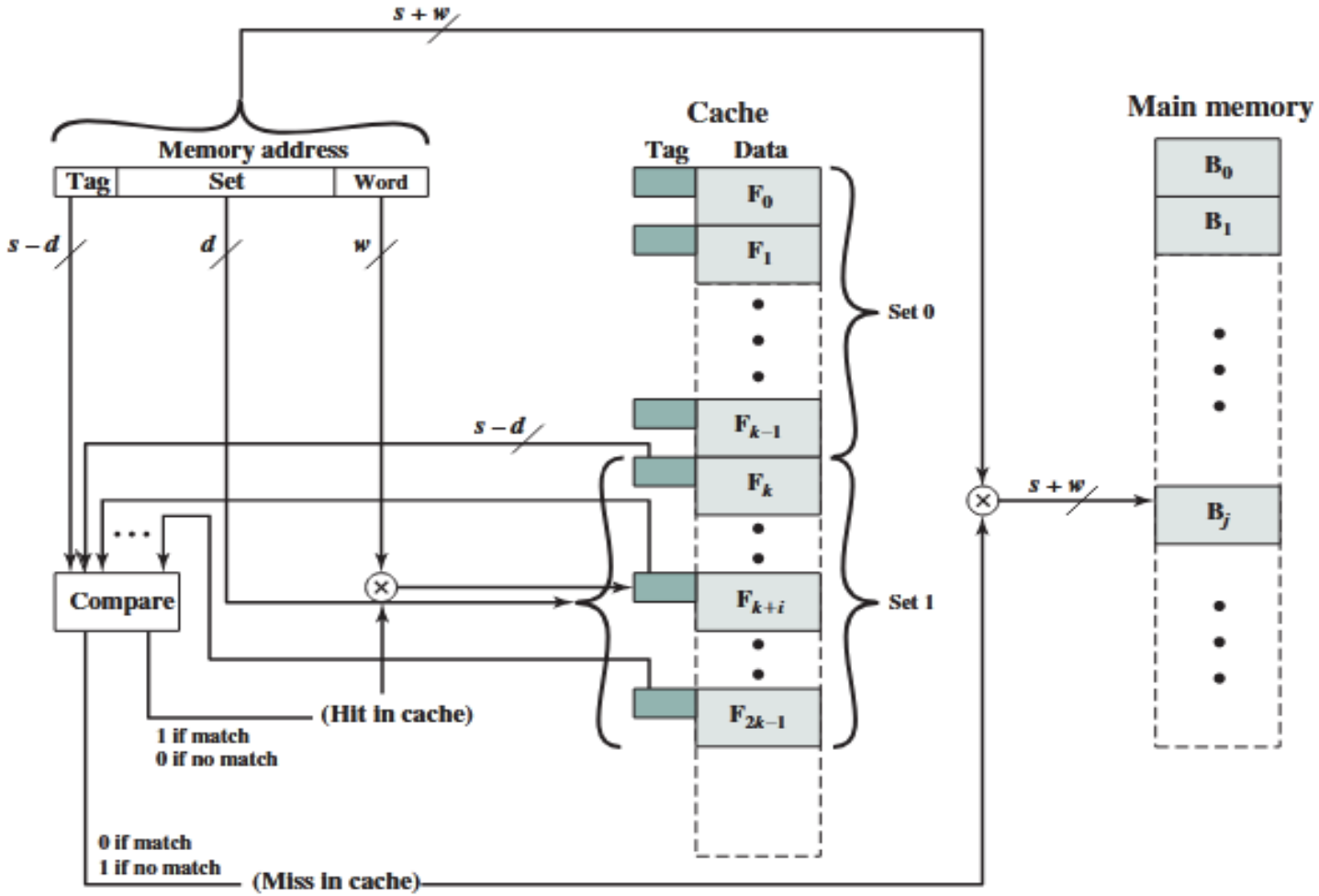


Figure 4.14 *k*-Way Set-Associative Cache Organization

Error Correction:

- Errors can be categorized as hard failures and soft errors.
- A hard failure is a physical defect(خلل). Hard errors can be caused by **harsh environmental abuse, manufacturing defects, and wear.**
- A soft error that **alters the contents of one or more memory cells without damaging the memory.** Soft errors can be caused by **power supply problems or alpha particles(جسيمات ألفا.).**
- Both hard and soft errors are clearly **undesirable**, and most modern main memory systems include **logic** for both **detecting** and correcting errors.

The comparison yields one of three results:

- ❖ No errors are detected.
- ❖ An error is detected, and it is possible to correct the error.
- ❖ An error is detected, but it is not possible to correct it.

Error Correction (cont'd) :

– Figure 4 illustrates in general terms how the process is carried out

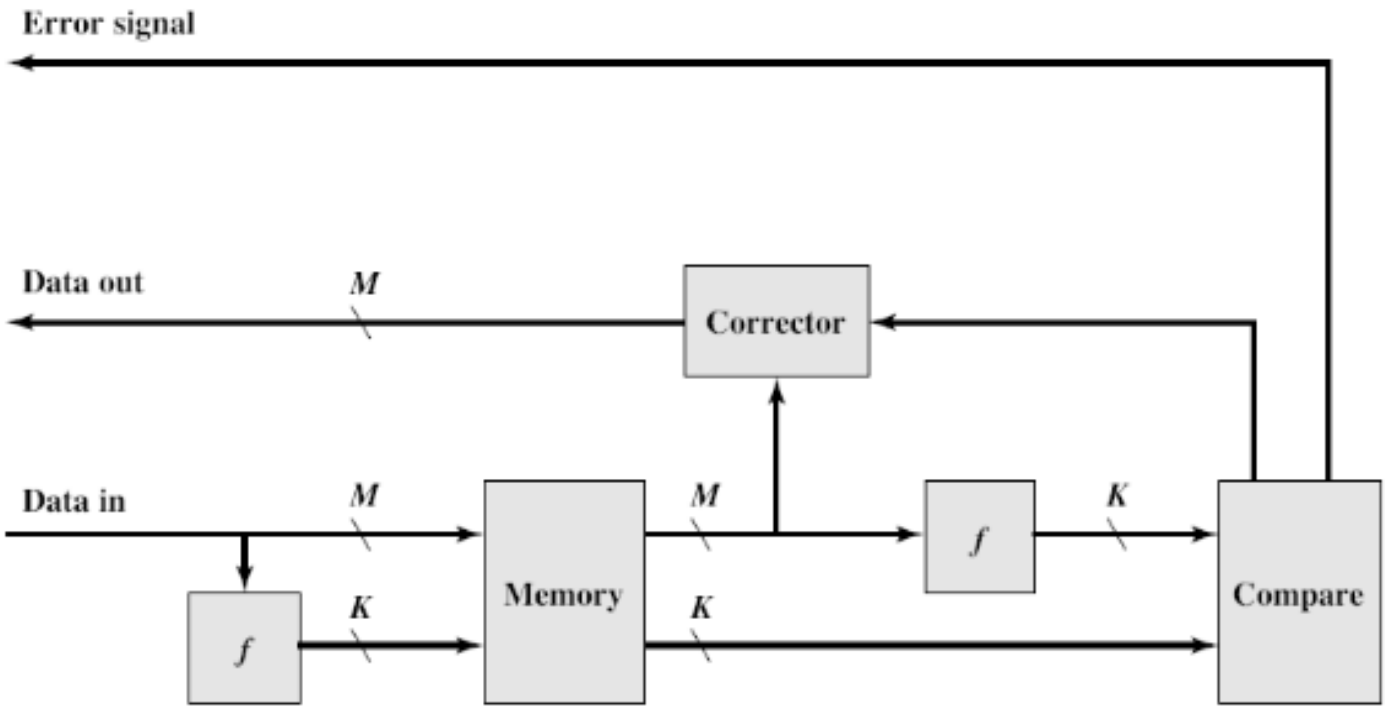


Figure 4 Error-Correcting Code Function

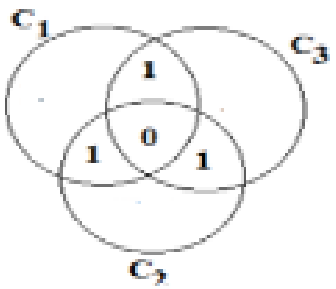
• Hamming Error-Correcting Code:

- The simplest of the error-correcting codes is the Hamming code that uses Venn diagrams that use of this code on 4-bit words (M=4) . With three intersecting circles.
- Example:

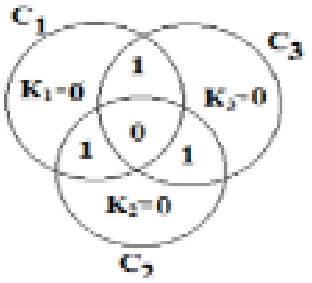
1) Use Hamming error-correcting codes on 4-bit data in (1101) and data out with error (1100)?

Solution: *1)*

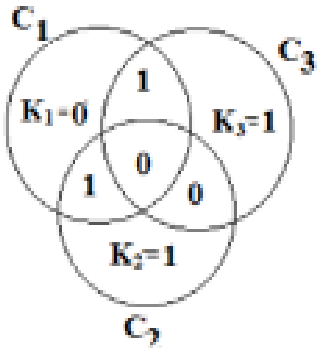
Data in



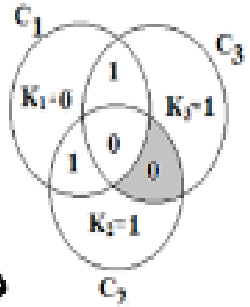
K: Parity code
K= 1 *if number of 1's is odd*
K= 0 *if number of 1's is even*



Data out with error



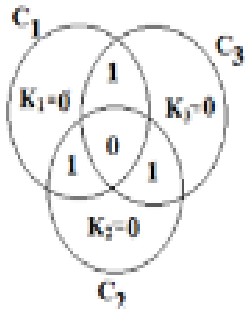
Detect →



Circle:	C_1	C_2	C_3
Code :	K_1	K_2	K_3
Old Code:	0	0	0
New code:	0	1	1

⇨ *Error bit = C2∩C3*

Correct



- **Error Correction (cont'd) :**

Solution: 2)