Chapter 5 Memory System Design Principles & Modern Memory Technology

Contents of this chapter

- Memory systems characteristics
- Useful properties in the memory systems
- Memory Capacity Planning for hierarchical memory systems
- Working Principles of Cache-based systems, Basic Read/Write strategies
- Memory mapping techniques
- Memory block Replacement techniques
- Working Set model & Memory allocation techniques
- Shared Memory Organizations
- Cache Coherence in Multi-core/Multi-CPUs & Protocols to resolve

ANNEXURE – Self-Reading on basic memory operations & VM

Memory Systems Characteristics

The memory technology at each level is characterized by the following five parameters.

1. Access time (t_i)

2. Memory size (s_i)

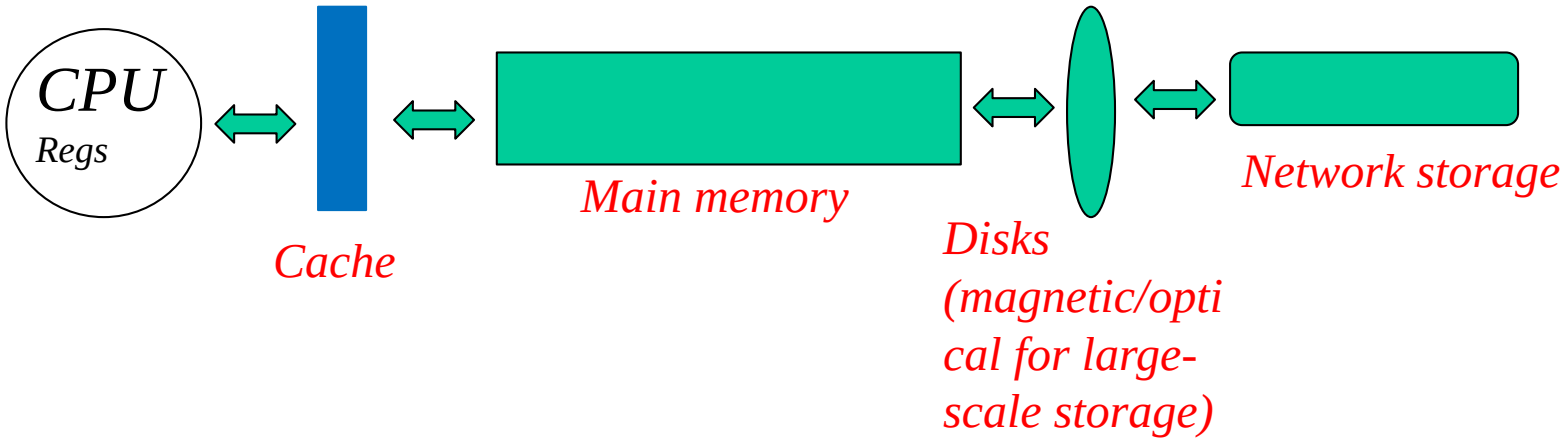
3. Cost per byte (c_i)

4. Transfer bandwidth (b_i)

5. Unit of transfer/granularity (x_i)

In general,

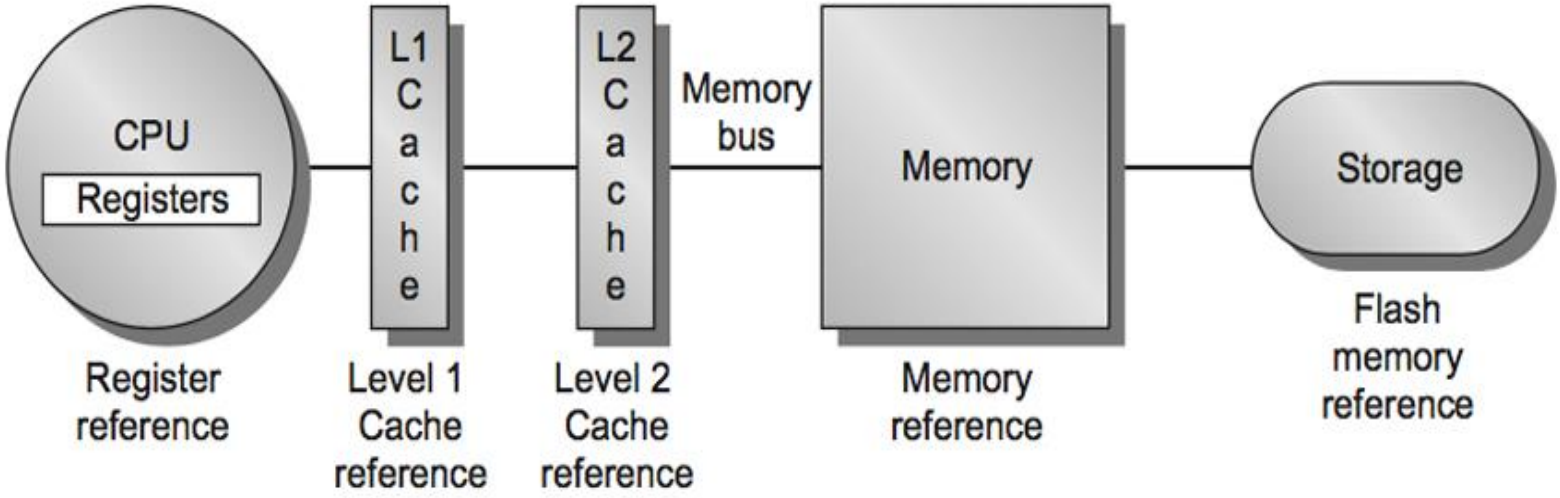
$t_{i-1} < t_i$, $s_{i-1} < s_i$, $c_{i-1} > c_i$, $b_{i-1} > b_i$, $x_{i-1} < x_i$, for all i=1,2,3,4.



Some useful properties in the design of memory technology

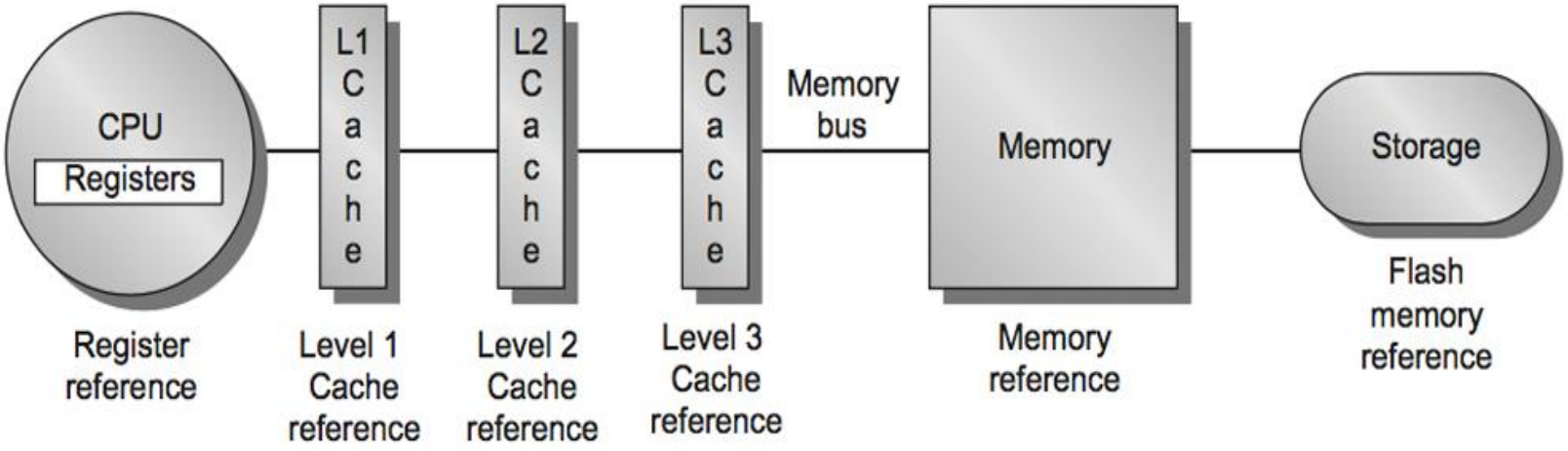
1. Inclusion 2. Coherence 3. Locality of References

Memory Hierarchy – Mobile Devices



Memory hierarchy for a personal mobile device

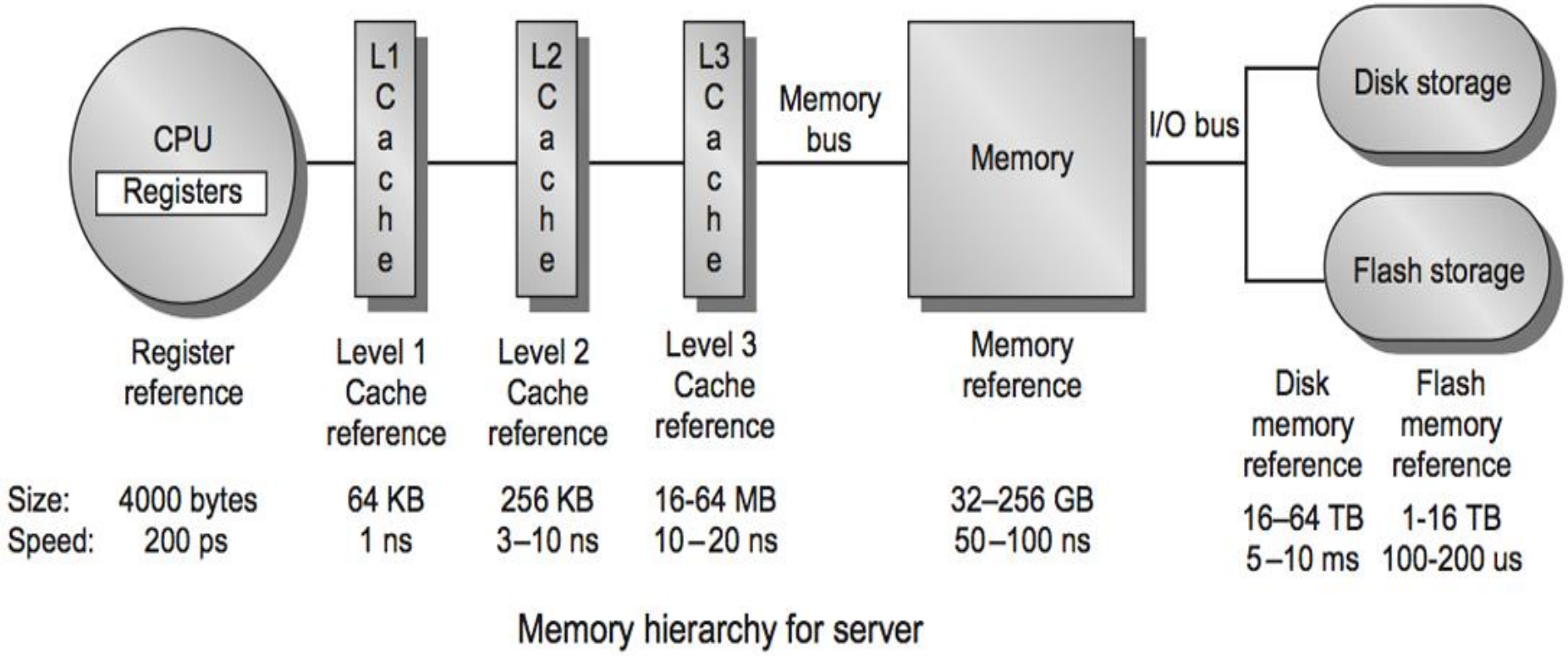
Memory Hierarchy – Typical laptop



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

Memory hierarchy for a laptop or a desktop

Memory Hierarchy – Typical Servers



Memory hierarchy for server

Let the memory hierarchy be denoted as $M_1, M_2, \ldots, M_n$ and consider the cache memory M_1, which directly communicates to CPU registers. The outermost level M_n contains all the information, and this level forms the virtual address space of memory hierarchy effectively.

Inclusion Property: This property is stated as

M_i *is a* ***proper subset*** *of* M_{i+1}, for all i=1,2,…,n-1.

During processing subsets of M_i are copied to M_{i-1} and so on. If an information is found in level i, then the information can also be found in all upper levels i+1 to n.

Information transfer between CPU and cache is in terms of words (typically 4 or 8 bytes depending on the word length of the machine).

Pages are the fundamental units that are transferred between MM and disks. Data transfer between disk and the tape units are in terms of file segments.

Coherence Property: This property emphasizes the need for the copies of same data to have same information at the levels wherever the data is currently residing. This is also referred to as *consistency* property. If a word is modified in the cache, then it must be updated at all levels.

Two strategies are followed to update, in general.

1. Write through: This demands immediate updates in a higher level memory if a word is modified in the immediate lower level memory.

2. Write back: This method delays the update in a higher level memory until the word being modified is replaced or removed from the immediate lower level memory.

LRU replacement policy will be studied in detail later in this chapter

Locality of references : The memory hierarchy basically originated by studying the program behaviour extensively. The behaviour refers to the “pattern” of access and the frequency of use of a page/word in the memory.

The access pattern tends to be clustered in certain regions in *time, space, and ordering*.

90-10 rule by Hennessy and Patterson (1990) states that a typical program may spend 90% of its execution time on only 10 % of the code such as the innermost loop of a nested looping operations.

Memory reference patterns are caused by the following locality properties:

1. Temporal: Recently referenced items are likely to be referenced in the near future

2. Spatial: Refers to the tendency for a process to access the items whose addresses are near to one another.

3. Sequential: In typical programs, execution of instructions follow an order unless any branch instructions are met.

The *sequentiality also contributes to spatial locality* as sequentially coded instructions and array elements *are often stored in adjacent locations*. Each type of locality affects the design.

The *temporal locality leads to LRU algorithm*; spatial locality helps in determining the size of the data units to be transferred between memory units; and the *sequential locality helps to determine the grain size for optimal performance*.

Working sets : The subset of pages or addresses referenced within a given window of time is referred to as WSs.

Vector/Matrix Storage & Accessibility

- Taking advantage of *Locality of Reference* property
- C has row-major storage for arrays (A1,1] if followed by A[1,2])

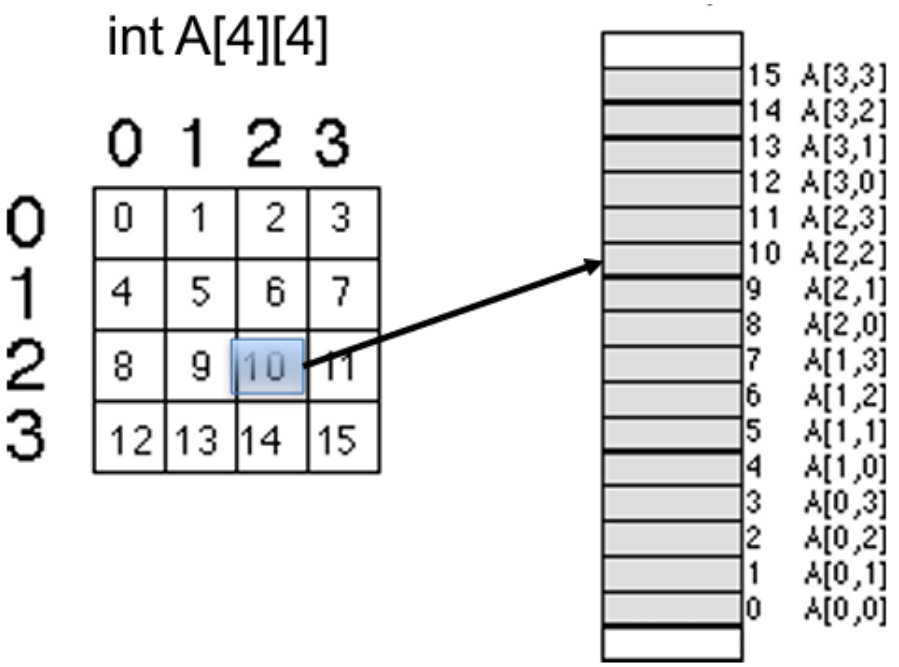
- Iterating Columns in a given row:

```
for(i=0;i<4;i++)
    sum += A[0][i]
```

- *Iterating rows in a given column:*

```
for(i=0;i<4;i++)
    sum += A[i][0]
```

Stride-4 jump!



The ability to examine code and gain a qualitative understanding of its locality is an essential skill for a professional programmer.

Does the following function offer a good locality?

```
int sumarrayrows(int a[M][N]){
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Does the following function offer a good locality?

```
int sumarraycols(int a[M][N]){
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Note that WSs are function of time, however, maintains some sort of continuity. The time window size is often crucial in the design of the size of the cache.

Memory Capacity Planning

The performance is determined by *effective access time* T_{eff} to any level in the hierarchy.

Hit ratios: When a memory M_i is accessed and if the desired word is found, it is referred to as a *hit*, otherwise it is considered as a *miss*.

The hit ratio (h_i) is the probability that a word/information will be found when accessed in M_i. It is a function of the characteristics of the adjacent levels. Obviously, the miss ratio is $1-h_i$.

The hit ratios at successive levels are a function of memory capacities, management policies, and program behaviour. In the analysis, successive hit ratios are *independent* random variables assuming values between 0 and 1.

Further, we assume that $h_0=0$ and $h_n=1$. This means that the CPU always access M_1 first and access to the outermost level is always a hit.

Access frequency at a level i is defined as

$$f_i = (1-h_1)(1-h_2)\ldots(1-h_{i-1})\, h_i$$

This is nothing but the probability of accessing the level M_i amidst (i-1) misses at the lower levels and hit at i. Note that $f_1 + f_2 + \ldots + f_n = 1$ *and* $f_1 = h_1$.

Due to the *locality property*, the access frequencies decrease rapidly from the lower levels, i.e., *access freq at level i is greater than i+1*. This means that the inner levels are accessed more often than the outer levels.

Effective Access Time is defined as

$$T_{eff} = h_1t_1+(1-h_1)h_2t_2+ (1-h_1)(1-h_2)h_3t_3+\ldots+ (1-h_1)(1-h_2)\ldots(1-h_{n-1})h_nt_n$$

Hierarchy Optimization The total cost of a memory hierarchy is estimated as follows:

$$C_{total} = c_1s_1+c_2s_2+\ldots+c_ns_n$$

Since $c_i>c_{i+1}$, we have to choose $s_i < s_{i+1}$. *The optimal design should result in a T_{eff} close to t_1 of M_1 and a total cost close to c_n of M_n*

In practice, it is difficult to achieve this objective. This problem is formulated as a linear programming problem. That is,

$$\textbf{min } T_{eff}, \text{ subject to: } s_i > 0;\ t_i > 0,\ i=1,\ldots,n \text{ and } C_{total} < C_0$$

<u>Refer to an example (next slide)</u>

Memory Hierarchy (Cont'd)…

Example: Consider a 3 level memory hierarchy for a small embedded electronic device that has limited computing power CPU with the following specifications for memory characteristics.

Memory Level	**Access time**	**Capacity**	**Cost/byte**
Cache	t_1 = 25nsecs	s_1 = 512Kbytes	c_1 = \$1.25
Main memory	t_2 = ?	s_2 = 32Mbytes	c_2 = \$0.2
External Disk	t_3 = 4 msecs	s_3 = ?	c_3 = \$0.0002

The design goal is to achieve an effective memory access time t = 10.04 micro-secs with a cache hit ratio of h_1 = 0.98 and a hit ratio of h_2 = 0.9 in the main memory. Also, the total budget given is upper bounded by \$15,000.00.

Memory Hierarchy Example (Cont'd)…

Solution:

The memory hierarchy cost can be computed as:

$C = c_1 s_1 + c_2 s_2 + c_3 s_3 \leq 15000$

The maximum capacity of the disk is thus obtained as s_3 = 39.8 GBytes without exceeding the budget.

Next, we want to choose a on device RAM memory after computing the access time. Using the expression for effective access time, we obtain,

$t = h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 \leq 10.04$

Substituting all the known parameters, we obtain t_2 = 903 nsecs

Q: What if you change the disk as a SSD (with t3 = 20 micro-secs)?

Classroom Discussions

Consider a 2 level memory hierarchy M1 and M2. Denote the hit ratio of M1 as h. Let c1 and c2 be the costs per kilobyte, s1 and s2 the memory capacities, and t1 and t2 the access times, resp.

(a) Under what conditions will the average cost of the entire memory system approach c2?

(b) What is the effective memory access time t_a of this hierarchical system?

(c) Let t2 = r. t1, where r denotes the speed ratio of the 2 memories M1 and M2. Let E = t1/ t_a be the access efficiency of the memory system. How will be the trend of E w.r.t r and h?

(d) What is the required hit ratio h to make E > 95%, if r = 100?

Memory Hierarchy Example (Cont'd)…

Some points to think!

Q1: Suppose one wants to double the main memory capacity at the expense of reducing the disk capacity under the same constraints. This change will / will not affect the cache hit ratio.

Q2: In the above Q1, it may increase the hit ratio in the main memory. (True/False)?

Q3: In the above Q1, the effective access time will be enhanced. (True/False)?

General working principle of cache based systems

When a read req is received from CPU, the contents of a block of memory words containing the location specified are transferred to the cache one word at a time.

Subsequently, when the program asks for any of the words from this block, the desired word is fetched directly from the cache.

However, suppose when a block occupying cache is not referenced for a long time, it is natural to push it back to the MM and fetch the required block.

Now, which block to replace is what is decided by replacement algorithms. And, where to place the incoming block in the cache is decided by the mapping function.

Note: *CPU does not need to know explicitly about the existence of cache. The CPU simply issues addresses that refer to locations in MM. The cache control circuitry determines whether the requested word currently exists in cache or not. If the desired word exists, the read/write operation will be done on the appropriate cache location. We refer to this case as a cache hit.*

Handling Read/Write Operations

In a write operation, the system can proceed in two ways.

(a). Write-through technique

(b). Write-back/Copy-back

Correct way of describing –

"Consider a write-through cache…"
"Assuming you are using a write-back policy…"

(a). Write-through: In this case, the cache and MM locations are simultaneously updated.

(b). Write-back: Update only the cache location and mark it as updated with an associated flag bit, often called as dirty or modified bit. The MM word is updated later, when the block containing the word is to be removed from the cache by a replacement algorithm.

Note: *(a) is simple, but results in unnecessary write operations in MM when cache is updated several times.*

(b) may also have unnecessary write operations, because when a cache block is written back to the memory, all the words of the block are written back, even if only a single word is modified in that block when it was in the cache.

What happens on a Read miss?

(a). When a read miss happens, the block containing the word is loaded into the cache and then the desired word is sent to the CPU.

(b). Load-through: Alternatively, this word may be sent to the CPU as soon as it is read from the MM. This is also referred to as early restart, as it reduces CPU's waiting time, but at the expense of additional circuitry.

What happens on a write-miss?

(a). Write-through: During a write-miss, the information is directly written into the MM location.

(b). Write-back: The desired word is brought back to the cache and updated.

Mapping Functions

There are three different mapping techniques that are followed in practice, depending on the sophistication available for implementation.

(a). Direct mapping

(b). Associative mapping

(c). Set-Associative mapping

Consider the following example that explains all these techniques.

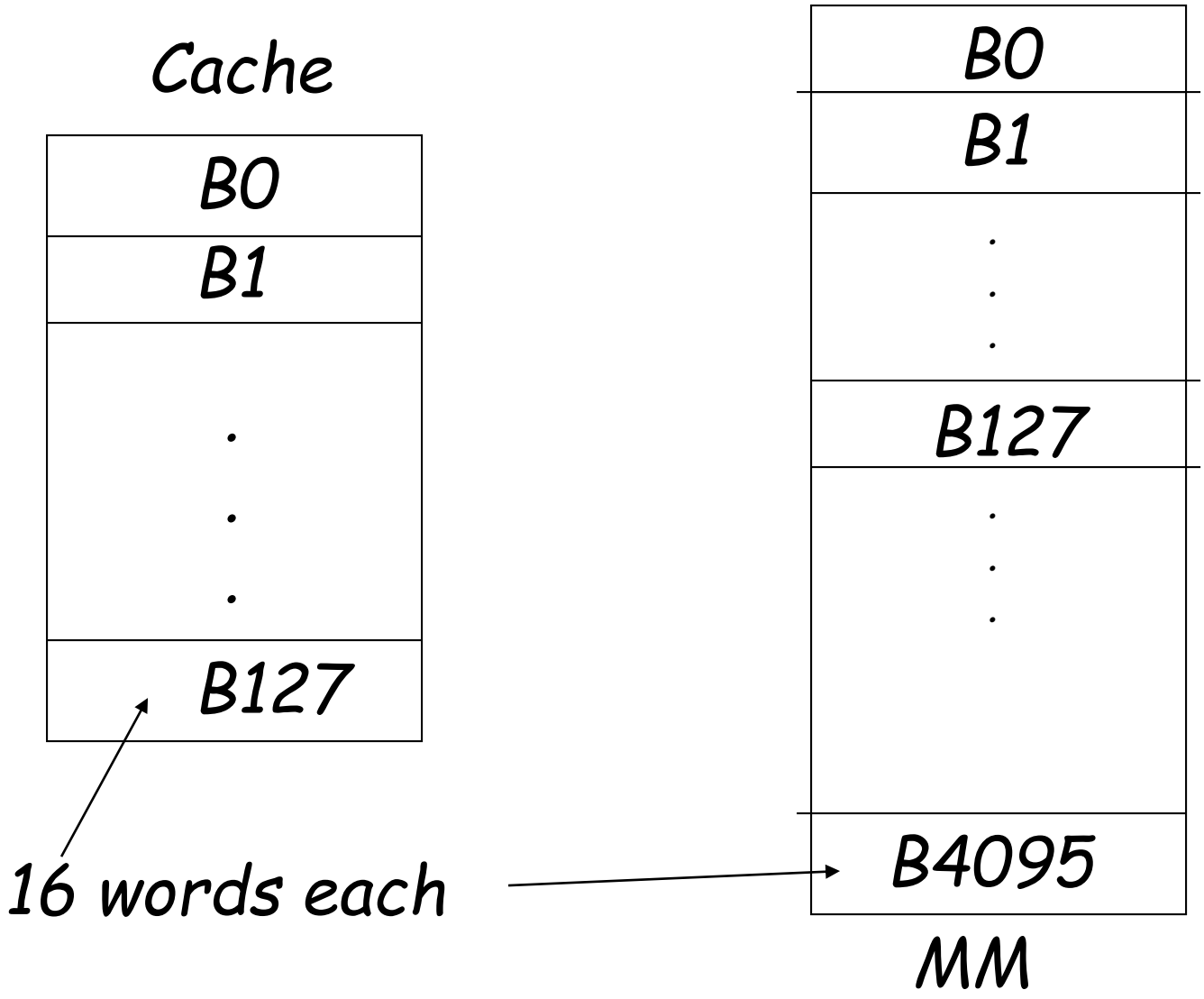
Let the cache consists of 128 blocks of 16 words each, for a total of 2048 (2K) words.

Assume that the MM is addressable by a 16-bit address . MM has 64K words, which we will view as 4K blocks of 16 words each.

(a) Direct mapping:

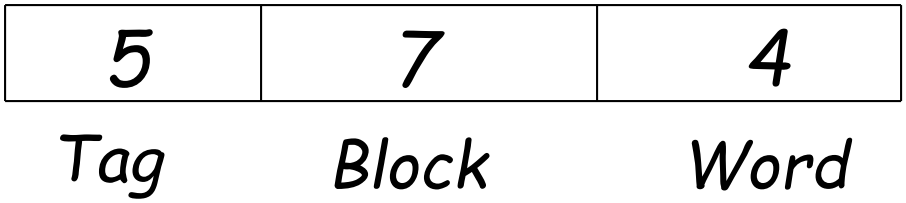
block j of MM -> block j modulo 128 of Cache

Thus, MM blocks 0,128,256,… -> block 0 of cache
MM blocks 1,129,257,… -> block 1 of cache,
and so on.


Cache
B0
B1
B127
16 words each
B0
B1
B127
B4095
MM

Note that even when the cache is not full, contention may arise for a location. In this case, the replacement algorithm is *trivial*.

Placement of a block in the cache is determined from the MM address generated. The MM address is divided into three fields as,



Note : Total of 16 bits; lower order -> select a word; middle order -> block position; high order -> which of the 32 blocks (4K/128 = 32) in MM is residing currently in cache

Note that the tag field in the above example is nothing but the higher order 5 bits of the word address .

These 5 bits are stored along with that block in the cache. So, when you reach the right block you can reach the word location, then using this tag field you can determine whether the word at this location is the wanted word or not, as the tag field is unique for each block.

Note: In all the mapping techniques, this is used. We ignore the size consumed by this tag field in the cache.

Classroom Discussions

Assume a computer has 32 bit addresses. Say, each block stores 16 words and a direct-mapped cache has 256 blocks.

In which block of the cache would we look for each of the following addresses? Addresses are given in hexadecimal
(i) 1A2BC012
(ii) FFFF00FF

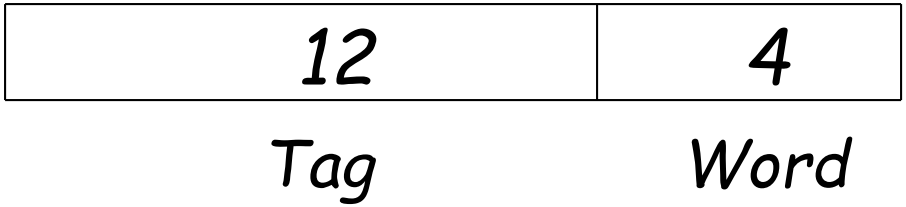
Solution:

Using 32 bit address, the last 4 bits denote the word in a block.

With 256 blocks in the cache, we need 8 bits to denote the block number. This would be the third to last and second to last hex digit. The rest are used as tag bits.
(i). this would be block 01, which is block 1
(ii). this would be 0F which is block 15

(b). Associative mapping:

In this technique, a block of MM can be placed anywhere in the cache, thus providing a flexibility in transferring the required block.



From the CPU generated address, the higher order 12 bits are stored along with the word in the cache, wherever space is available. When the request arrives the tag field is compared

for all the words in the cache to see the match. This technique gives a complete freedom in choosing the cache location, and hence the cache space is utilized more efficiently.

The replacement follows some of the standard techniques LRU, FIFO, etc.

Disadvantage: Search 128 blocks to match for a single tag, hence costly; parallel search schemes can be used.

(c). Set-Associative mapping:

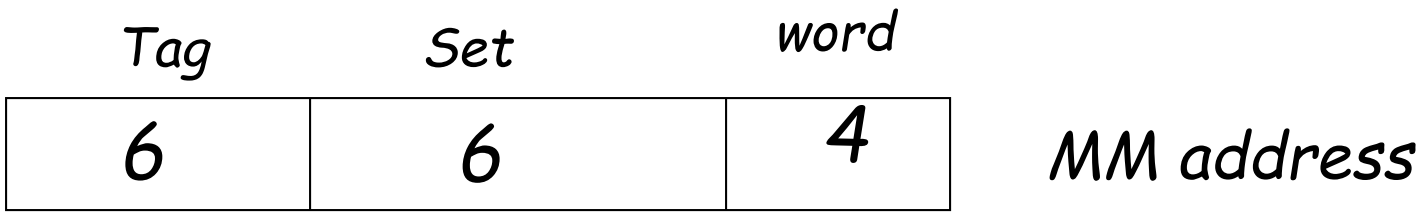
This is a combination of the previous techniques.

Here, blocks of cache are grouped into sets, and the mapping allows a block of the MM to reside in any block of a specific set.

Thus, the contention problem of the direct method is eased by having few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search procedure.

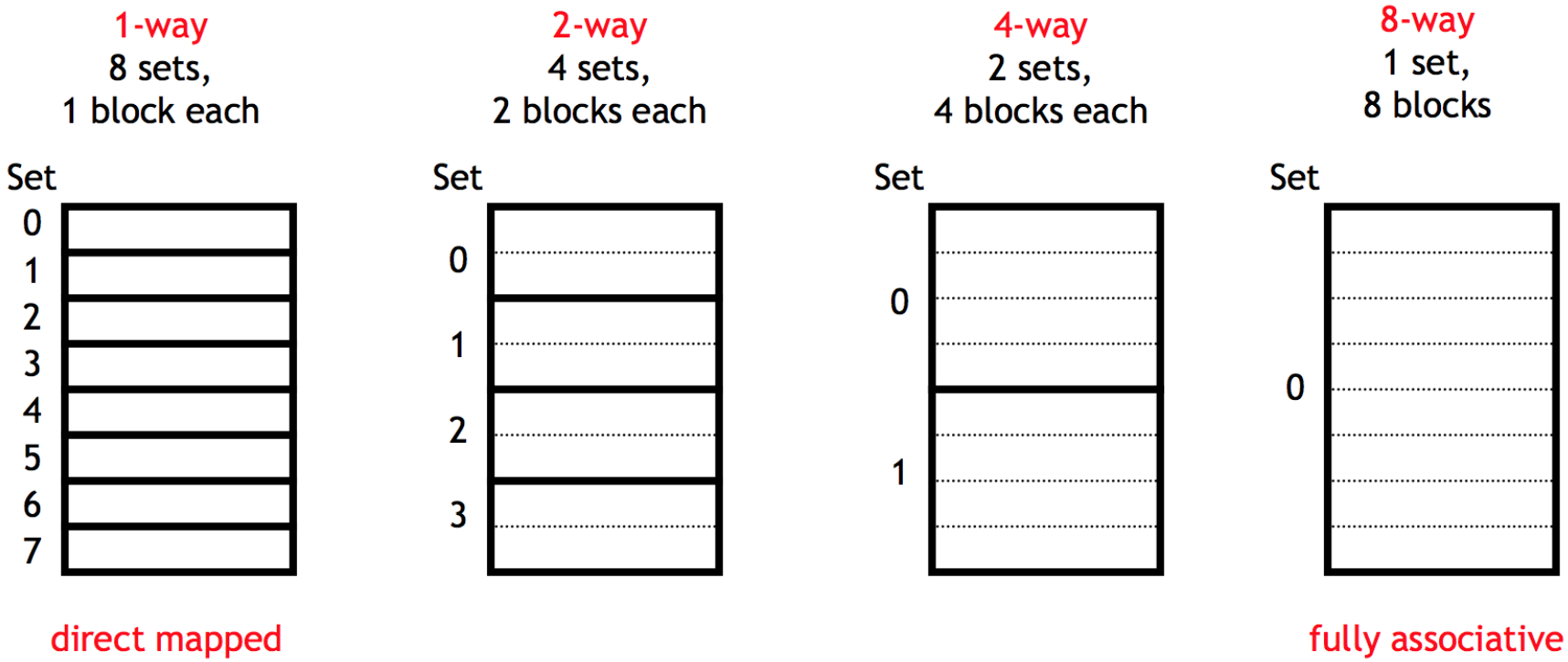
With our example, suppose if we allow two blocks per set in the cache. This means, the memory blocks 0,64,128,…,4032 map into cache set 0, and they can occupy either of the two block positions within the set.

With 128 cache blocks and 2 blocks per set we have 64 sets implying we need 6 bits to identify the right set and 4 bits for a word leaves 6 bits for the Tag field. Thus,



Memory Mapping Functions (Cont'd)...

From Direct Mapped to Fully Associative



Classroom Discussions

An IoT system with an embedded CPU has a 8 GByte on-board memory with 64 bit word sizes. Each block of memory stores 16 words. The computer has a direct-mapped cache of 128 blocks. The CPU uses word level addressing.
(a) What is the address format?
(b) If we change the cache to a 4-way set associative cache, what is the new address format?

Section A type Q

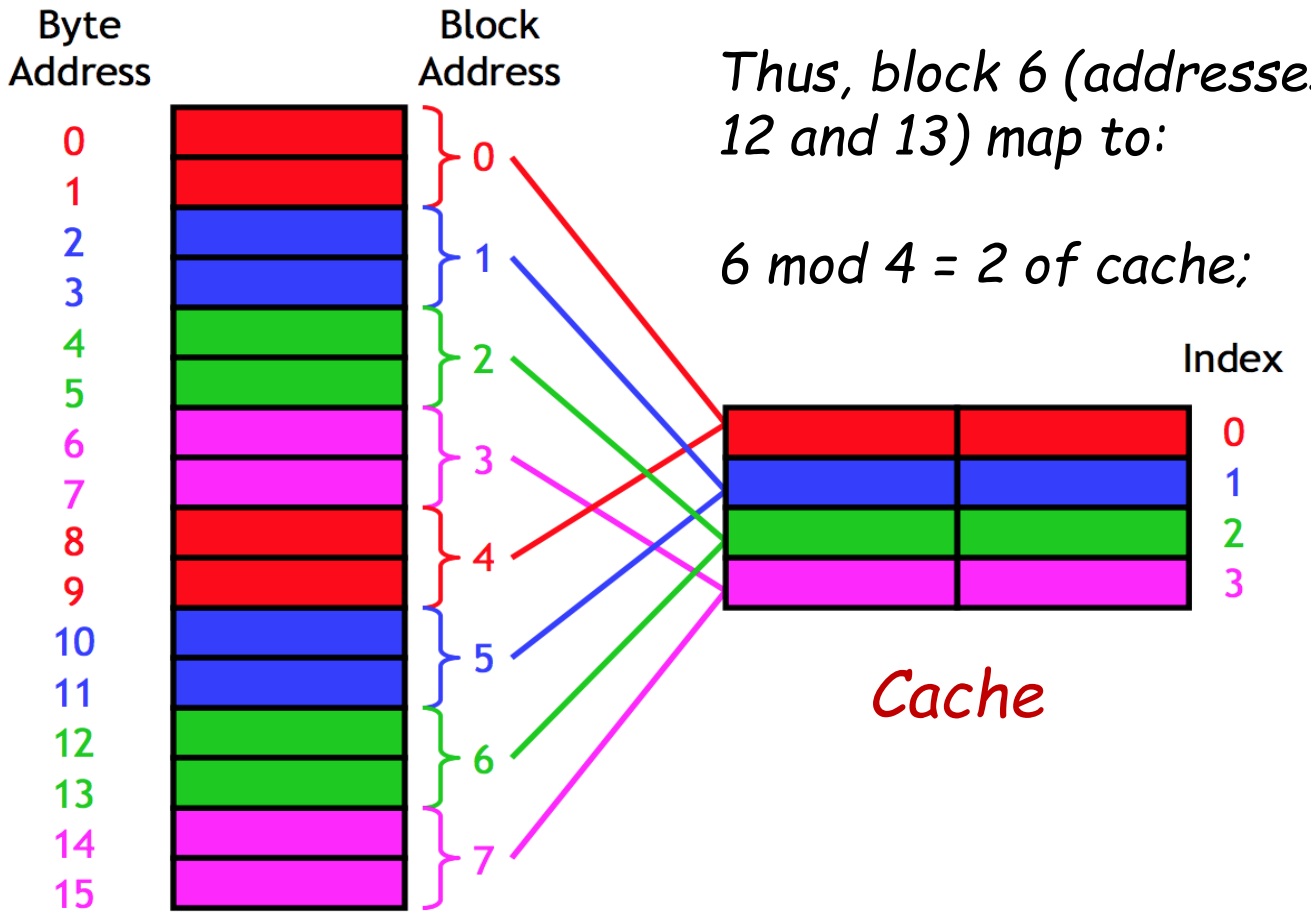
Solution: (Verify!)

(a) Address format is 19 – 7 – 4
(b) Address format is 21 – 5 – 4

Direct mapped cache – Identifying Valid & Offset bits

Applying Spatial Property:

Let each cache block has 2 bytes;



Thus, block 6 (addresses 12 and 13) map to:

6 mod 4 = 2 of cache;

Cache

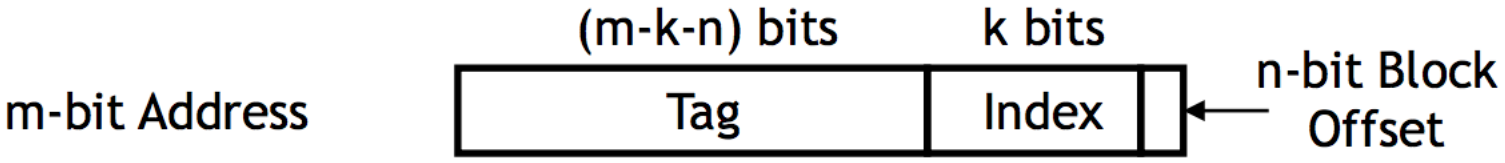
Direct mapped cache… (cont'd)

So, within the cache block where do the bytes go?

Lower byte (blk 12) – Lower addr;
Higher byte (blk 13) – higher addr

Depending on the CPU, Little Endian convention may be used – Lower byte to lower address and higher byte to higher address;

In general, Cache: 2^k blocks with each block containing 2^n bytes, we have:

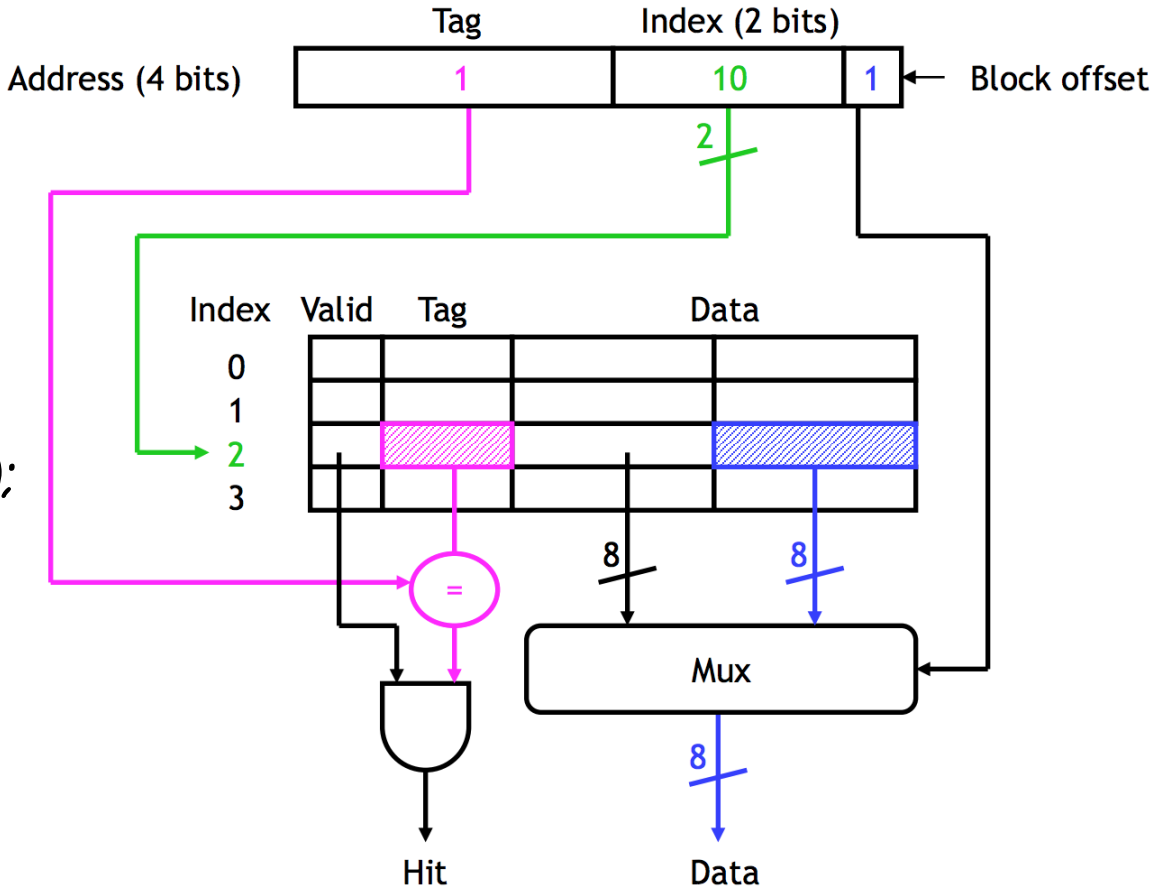


Direct mapped cache… (cont'd)

4 bit address

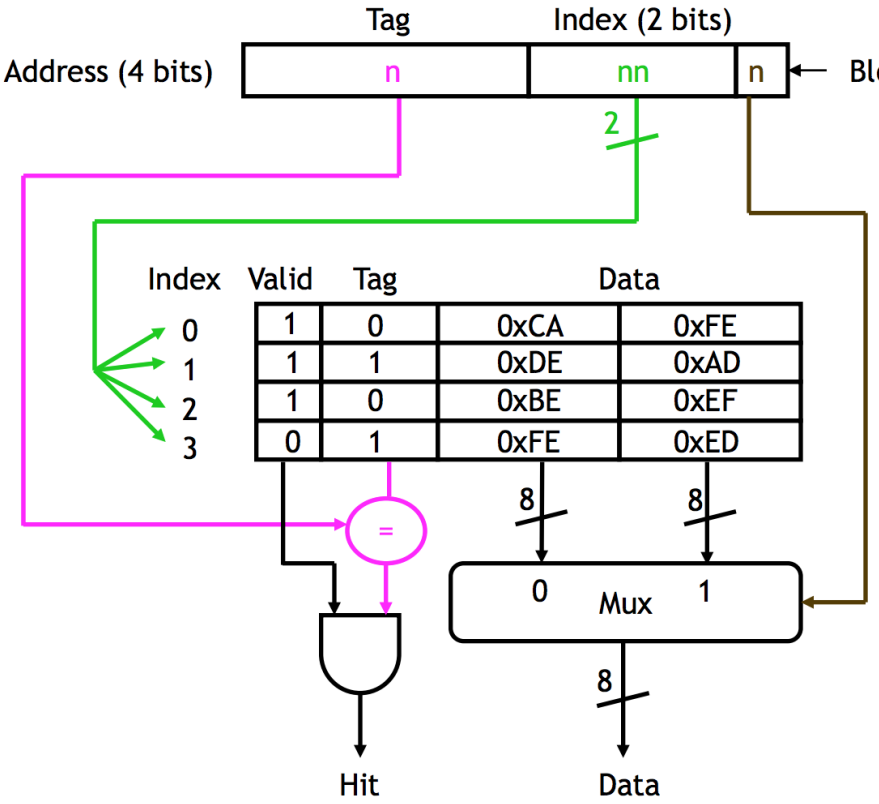
Block 6 comprising Addresses 12 & 13 Map to Block 2 (10) of cache;

Thus, index: 10 (2 bits); Within that block we have 2 bytes – hence we need 1 bit offset to locate the byte; Rest of the bits – 4 – 2 -1 = 1 bit serves as a Tag bit.



Direct mapped cache… (cont'd)

Try this exercise!



For the addresses below, what byte is read from the cache (or is there a miss)?

1010

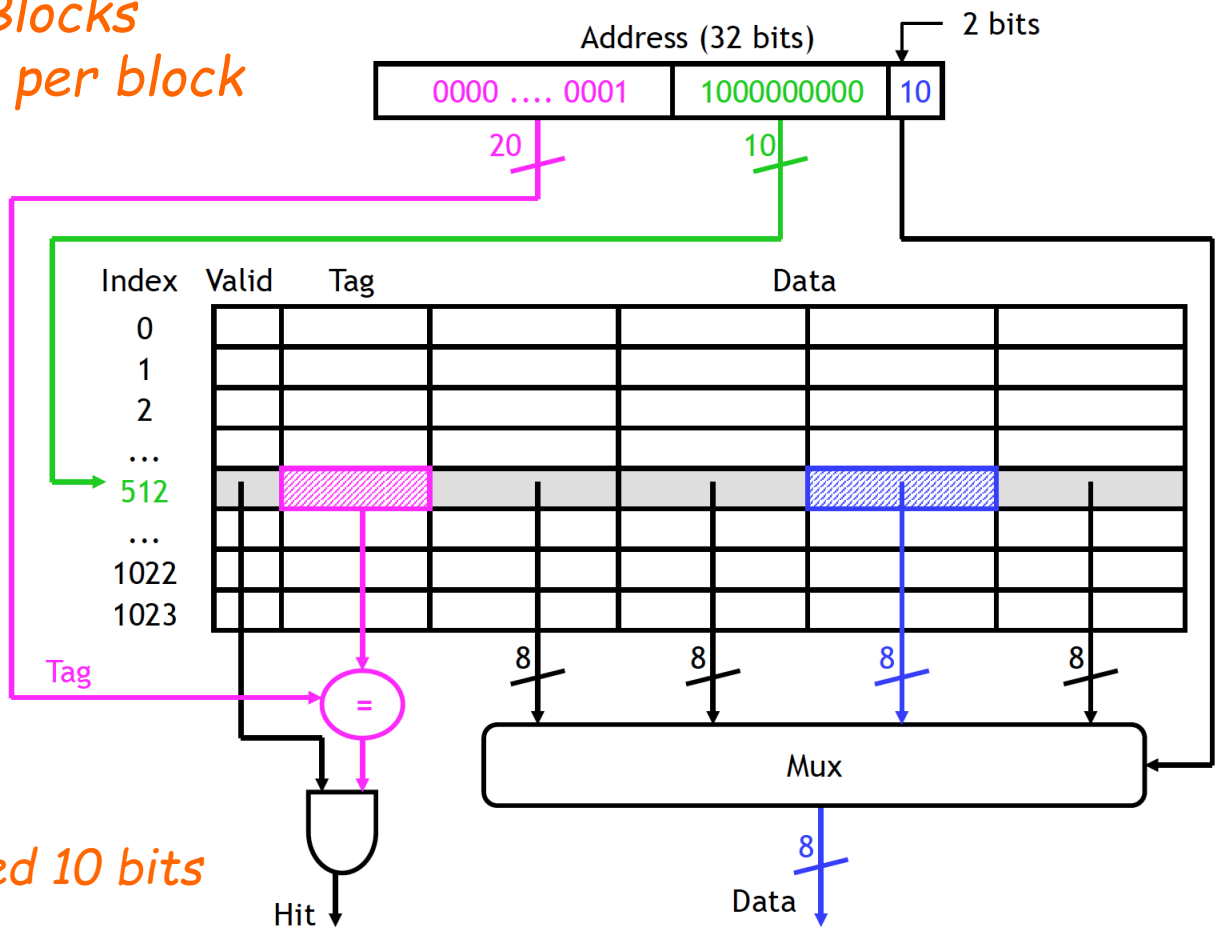
1110

0001

1101

Larger Cache Example

- MM: 32 bit address
- 1024 Cache Blocks with 4 bytes per block



1024 Blocks - Need 10 bits for a block;
4 Bytes per block – need 2 bits;

Remarks:

How do I prevent any stale data residing in the cache, if I do not use write-through method?

We need a special bit referred to as valid bit to indicate whether the data is stale or not.

When the power is switched on, all these valid bits are set to 0. The valid bit is set to 1 when a block comes to the cache for the first time. Whenever the MM is updated by a source that bypasses cache, a check is made to determine whether the block being used is currently in the cache. If it is, the valid bit is cleared to 0. This solves the problem.

Replacement Algorithms

In the case of direct mapping technique, the position of each block is predetermined, and hence there is no replacement strategy.

However, in the other two cases there is some flexibility.

Problem: When a new page is brought to the cache for the first time, and if all the slots are occupied, the problem is who has to be thrown out?

The cache controller unit must decide on this quickly. Note that the very purpose of the cache is to retain certain blocks as they are likely to be referenced in the "near future". However, it is not easy to decide on "how long to hold a block".

The locality of reference property gives a clue.

It is makes sense to overwrite a block that resided in the cache for a long time without being referenced. This block is referred to as the least recently used (LRU) block and the strategy is referred to as the LRU algorithm.

Workings of LRU – Example

Page access follows the sequence indicated; Assume cache has 4 slots, initially kept empty;

time

7
0
1
2
0
3
0
4
2
3
0
3
2

Follow the access pattern and identify all LRU Pages.

*Initially pages 7 0 1 2 are allocated to the empty slots > **4 Page faults***

*Page 3 request arrives – Replace 7 (LRU page); So, we have **1 Page fault** here;*

*Page 0 is already available; Page 4 req arrives; Replace Page 1 (LRU page); So, we have **1 Page fault** here;*

Total of 6 page faults

These came and waited for a long time without being referenced

LFU Replacement algorithm

LFU keeps a list of all the pages referenced in the cache and how many times they have been referenced in the past – frequency count is done!

Once the cache becomes full it replaces the page that has been referenced the fewest – lowest counter value, with the new one; In case of a tie, it can choose arbitrarily.

Consider the same given Seq; Determine the LFU pages;
7 0 1 2 ***0*** *3 0 4* ***2*** *3 0* 3 2

Classroom Discussions

		Pg-count	
7			
0			
1			
2	*4 faults until here;*	*7-1, 0-1, 1-1,2-1*	
0	*4 faults until here;*	*7-1, 0-2, 1-1,2-1*	
3	*5 faults until here;*	*3-1, 0-1, 1-1,2-1;*	*7 replaced*
0	*5 faults until here;*	*3-1, 0-3, 1-1,2-1*	
4	*6 faults until here;*	*3-1, 0-3, 1-1,4-1;*	*2 replaced*
2	*7 faults until here;*	*2-1, 0-3, 1-1,4-1;*	*3 replaced*
3	*8 faults until here;*	*2-1, 0-3, 1-1,3-1;*	*4 replaced*
0			
3			
2			

Total faults: 8

Optimal algorithm

A Clairvoyant strategy!!! This is where the system can look at all the references that will happen in the future. The page that is then chosen to be replaced is the one that is either not being referenced again, or is being referenced again the farthest into the future compared to the other in the cache.

Consider the same given Seq; Determine the # of faults;

7 0 1 2 **0** 3 0 4 **2** 3 0 3 2

4 faults

Page fault;
Replace 7 – obvious choice!
We have:
3 0 1 2 in the cache

Page fault;
Replace 1;
We have:
3 0 4 2

Total faults: 6

First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue; It keeps the oldest page in the front of the queue. So, track the head-of-the-queue in this scheme.

When a page needs to be replaced, the page in the front of the queue is selected for removal.

Page sequence:
a, b, c, d, c, a, d, b, e, b, a, b, c, d

Size of the cache memory be 4 page frames.

Compute & compare the total number of faults Using FIFO, LRU, & LFU.

Belady's Anomaly

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames!

This is especially true for FIFO!

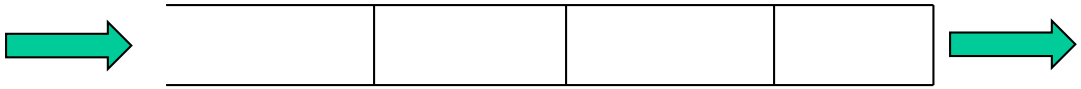
Consider a sequence d,c,b,a,d,c,e,d,c,b,a,e;
With 3 page frames, we get 9 total page faults, but if we increase page frames to 4, we get 10 page faults!

Why this happens? Work out the solution and realize the mechanics:

3-page frames case: 9 faults

3rd element	2nd element	Head of the Q	miss/hit	time
		d	m	1
	c	d	m	2
b	c	d	m	3
a	b	c	m	4
d	a	b	m	5
c	d	a	m	6
e	c	d	m	7
		d	h	8
	c		h	9
b	e	c	m	10
a	b	e	m	11
		e	h	12

d,c,b,a,d,c,e,d,c,b,a,e



4-page frames case: 10 faults

4th element	3rd element	2nd element	Head of the Q	miss/hit
			d	m
		c	d	m
	b	c	d	m
a	b	c	d	m
a	b	c	d	h
a	b	c	d	h
e	a	b	c	m
d	e	a	b	m
c	d	e	a	m
b	c	d	e	m
a	b	c	d	m
e	a	b	c	m

Compare with 3 page frame case at t=7

Compare with 3 page frame case at t=11

d,c,b,a,d,c,e,d,c,b,a,e

The reason that when using FIFO, increasing the number of pages can increase the fault rate in some access patterns, is because when we have more pages, recently requested pages can remain at the bottom of the FIFO queue longer.

Bélády's Anomaly does not state anything about the general trend of fault rates with respect to cache size.

In summary: Bélády's Anomaly points out that it is possible to exploit the fact that larger cache sizes can cause items in the cache to be raised in the FIFO queue later than smaller cache sizes, in order to cause larger cache sizes to have a higher fault rate under a particular (and possibly rare) access pattern.

Working Set Model

In the beginning, processes are started up with none of their pages in the memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the OS to bring in the page containing the first instruction.

After a while, the process has most of the pages it wants and settles down to run with relatively fewer page faults. This strategy is called demand paging because the pages are loaded only on demand, not in advance.

The set of pages that a process is currently using is called its working set.

If the entire working set is in the memory (cache), then the process will run without any large number of page faults, until it moves into another execution phase.

A program causing page faults every few instructions is said to be thrashing.

In a time-sharing system, processes move frequently between disks and MM.

The question arises of what to do when a process is brought back again. Technically, nothing needs to be done! The process will just cause some page faults until its working set has been loaded.

However, the problem is that, having 20, 60, or even 100+ page faults every time a process is loaded is slow, and it also wastes considerable CPU time.

Therefore, many paging systems try to keep track of each process's working set, and make sure that it is in memory before letting the process run.

Local versus Global allocation policies

Original configuration

Pages: A0 A1 A2 A3 A4 A5 B0….B3…C1..C3
Age : 10 7 5 4 6 3 9 …. 2… 3 … 6

Question: Suppose process A gets a page fault. Should the replacement algorithm try to find the LRU page considering only the 6 pages currently allocated to A, or consider all the pages in the memory?

Note: Age -> hit count

Local replacement policy: This considers only A's pages, page with the lowest age value A5, and hence page A6 is replaced instead of A5 in its current place.

Global replacement policy: Here, the page with the lowest age value is chosen, regardless of the fact that whose page it is. In our example, it is page B3, and hence it is replaced with page A5.

Local algorithms correspond to assigning each process a fixed amount of memory

Global algorithms dynamically allocate page frames among the runnable processes. Thus, the number of pages assigned to each process varies in time.

In general, global algorithms work better, especially when the working set size can vary over the lifetime of a process.

If a local algorithm is used and when the working set grows, thrashing will result, even if there is enough room.

If the global algorithm is used, the system must continually decide on how many page frames (slots) to assign to each process. One way is to monitor the working set size as indicated the "aging bits", but there is no guarantee for a fault-free situation. *The working set size may change in microseconds, whereas the aging bits are a crude measure spread over a number of clock ticks.*

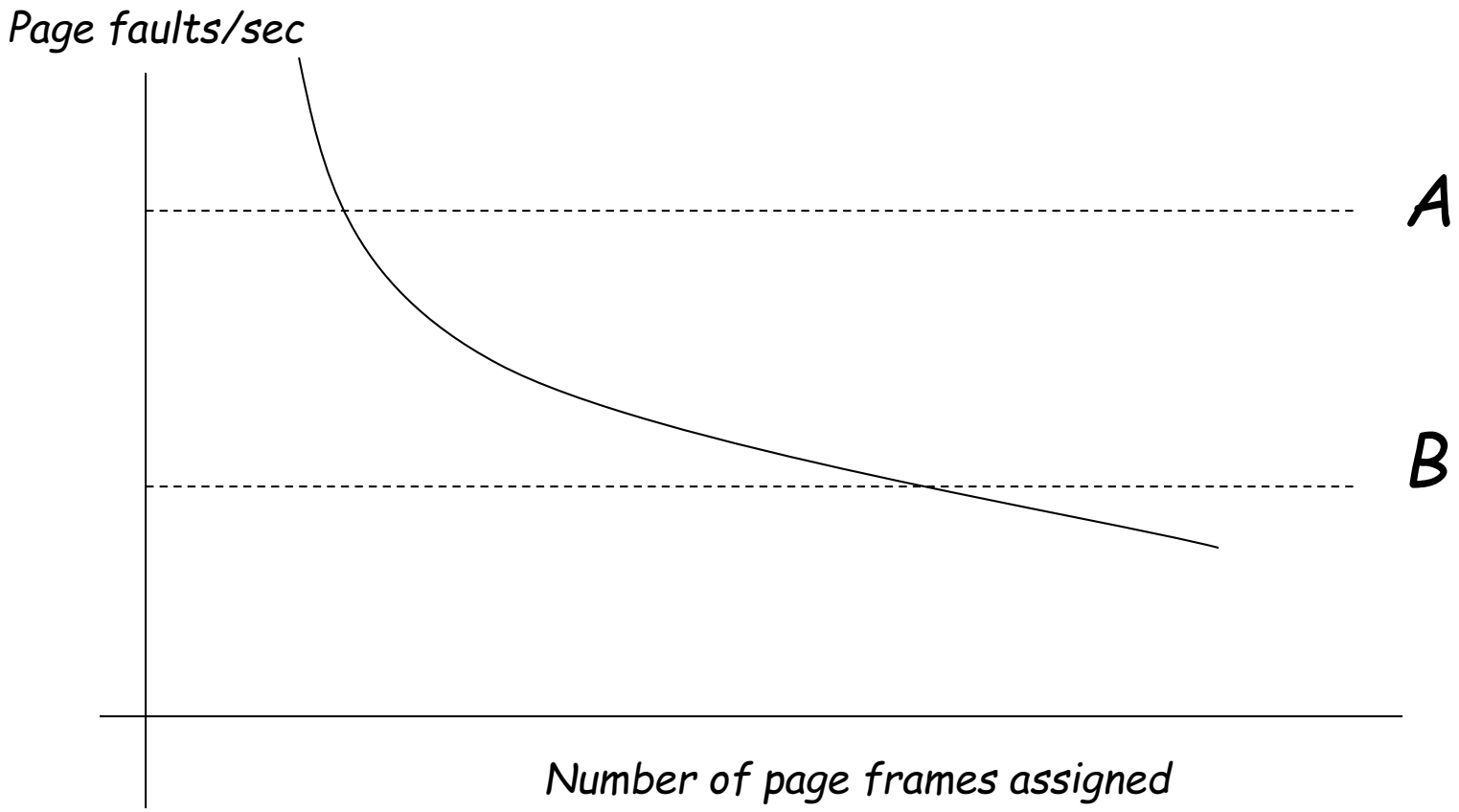
Equal sharing strategy: In this approach, the OS periodically determines the number of running processes and allocate each process an equal share.

Thus, if 475 frames (slots) are available currently and if 10 processes are running, 47 frames are allocated to each process and the remaining 5 frames are reserved for the situation when a page fault occurs.

Although this approach seems fair, it makes little sense to give equal share of memory to a 10K process and a 300K process.
-- Instead, pages can be allocated in proportion to each process's total size. That is, with a 300K process getting 30 times the allotment of a 10K process.

Note: On some machines, for example, a single instruction may need as many as 6 pages because the instruction, the source operand and the destination operand may all straddle page boundaries. In this case, with an allotment of say, only 5 pages, programs containing such instructions may not run at all!!

- For a large class of page replacement algorithms, including LRU it is known that the fault rate decreases as more and more pages are assigned.



The dashed line marked A corresponds to a page fault rate that is unacceptably high, so the faulting process is given more slots to reduce the fault rate.

The dashed line marked B corresponds to a page fault rate so slow that it can be concluded that the process has too much memory. In this case, the page frames may be taken away from it.

This approach is referred to as page fault frequency algorithm or simply PFF algorithm.

Thus, PFF tries to keep the paging rate within acceptable bounds.

Shared Memory Organizations

Main issue – Matching the speed of the CPU processing with data access from memory!

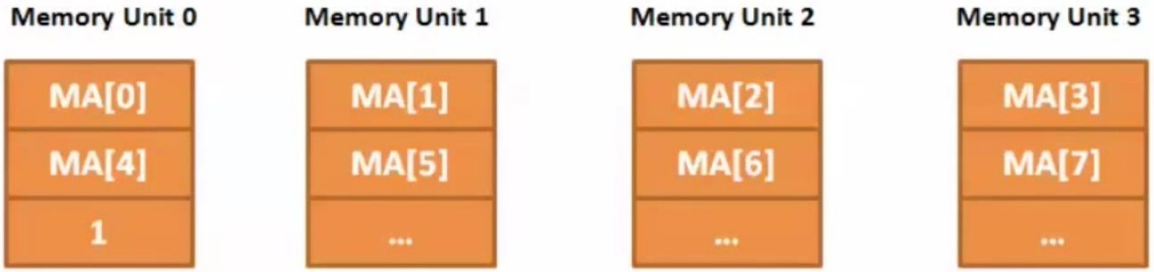
This is a bottleneck when working at higher speeds as memory system is unable to cope with CPU processing speeds.

- *Goal: Maximize the effective memory bandwidth so that more words can be accessed per unit time;*

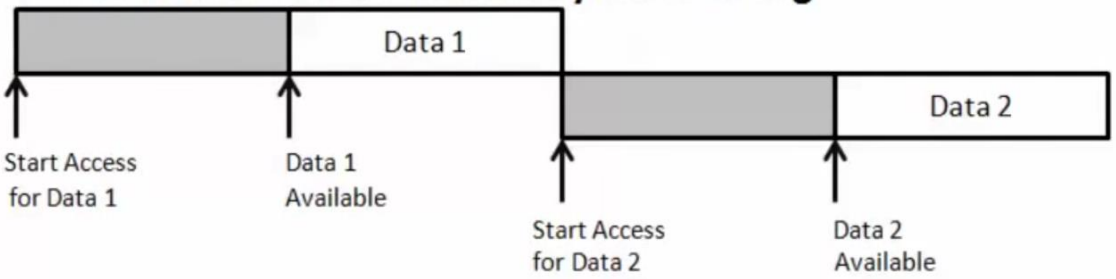
Matching the Memory bandwidth + bus bandwidth + processor bandwidth

Memory Interleaving

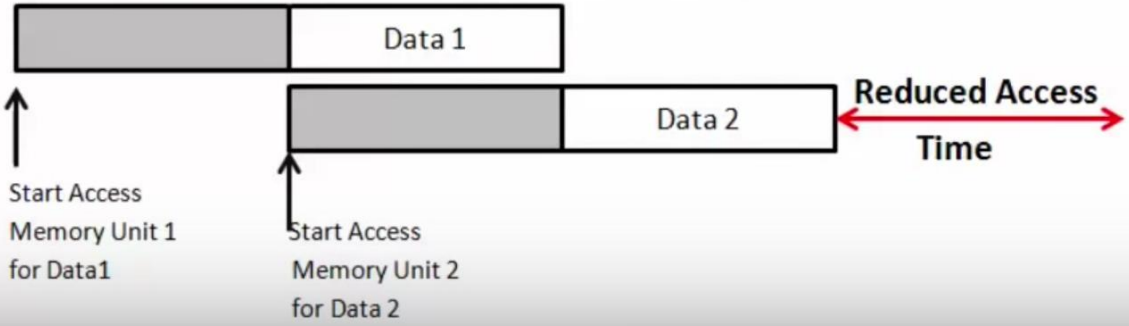
Technique to interleave successive memory addresses across multiple memory units



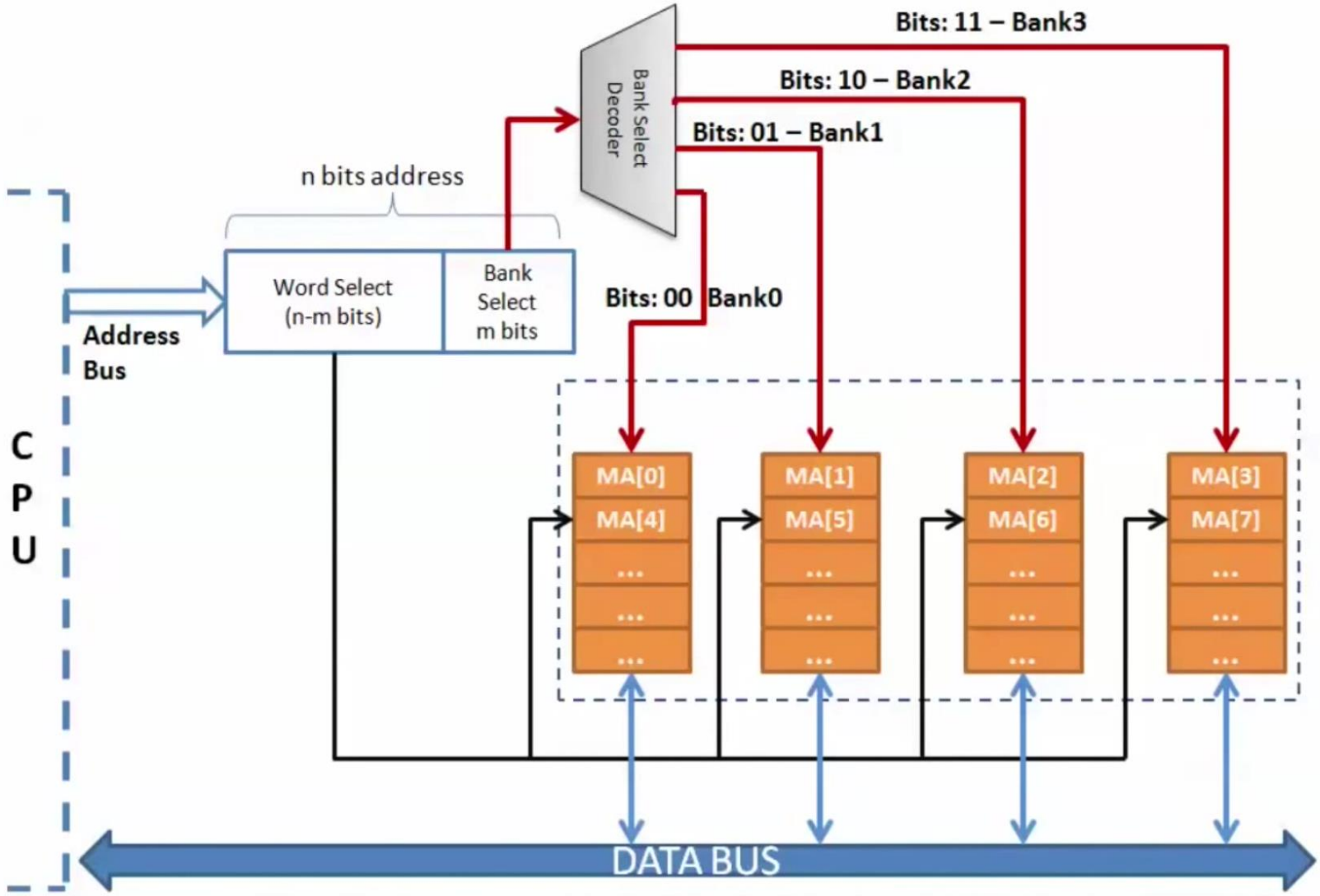
Access Pattern without Memory Interleaving



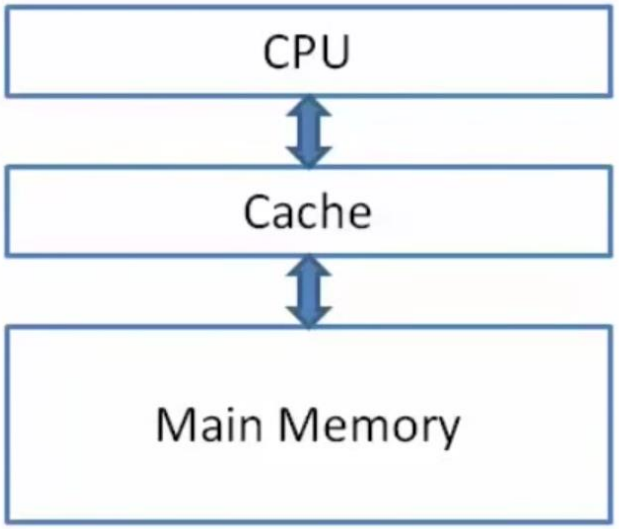
Access Pattern with Memory Interleaving



Abstract Architecture



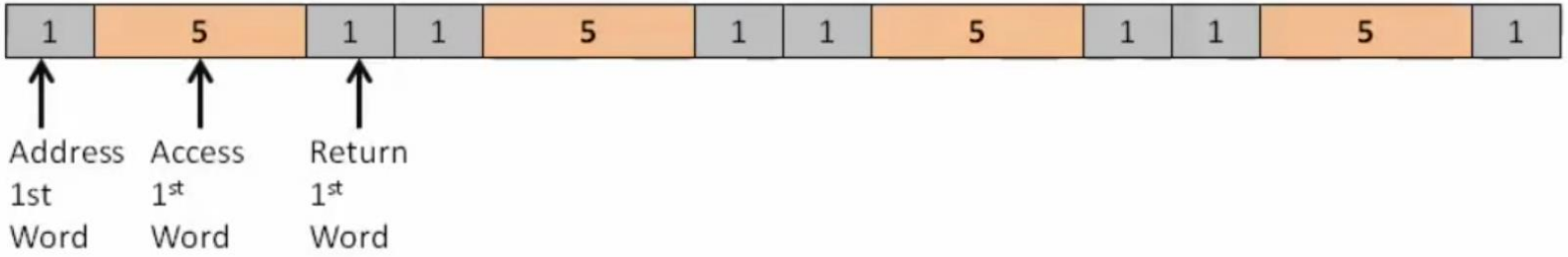
Non-Interleaved Memory Access with No Bank Division



4 word wide Cache

1 cycle to send address to DRAM
5 cycles for one word DRAM access
1 cycle to return data

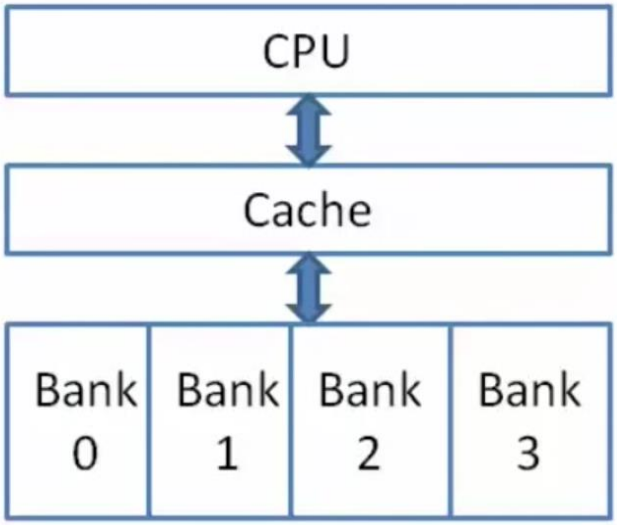
To Access 4 words of Data from Memory:



Time Taken to fetch 1 word from DRAM = 1 + 5 + 1 = 7 cycles

Time Taken to fetch 4 words from DRAM = 4 * (1 + 5 + 1) = **28 cycles**

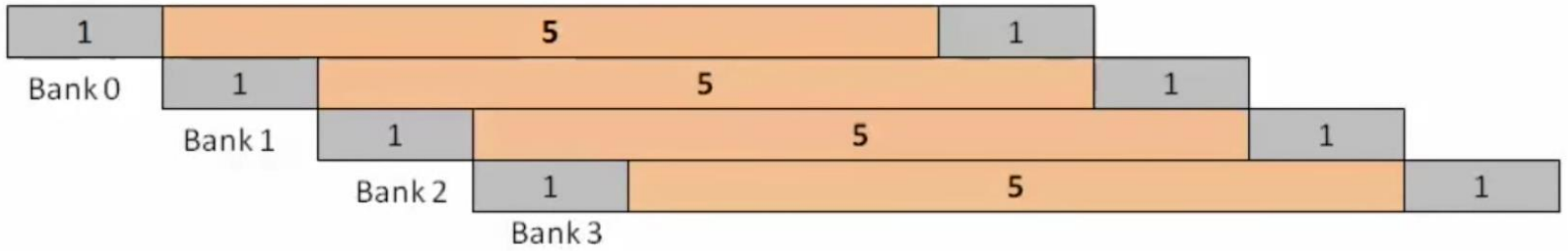
Interleaved Memory Access with 4 Banks



4 word wide Cache

1 cycle to send address to DRAM
5 cycles for one word DRAM access
1 cycle to return data

To Access 4 words of Data from Memory:



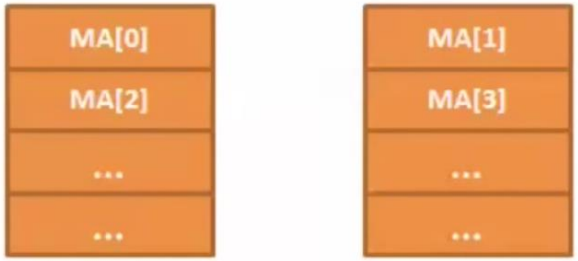
Time Taken to fetch 1 word from DRAM = 1 + 5 + 1 = 7 cycles

Time Taken to fetch 4 words from DRAM = (1 + 5 + 1) + (3*1) = **10 cycles**

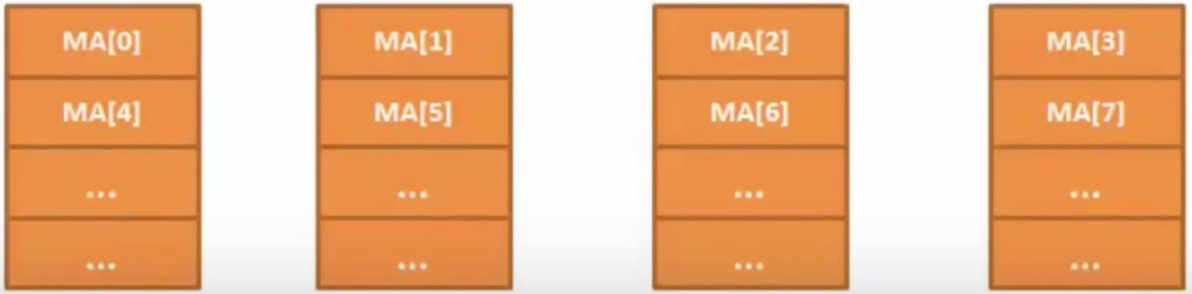
n – Way Memory Interleaving

The memory can be divided into multiple memory units. n – way interleaved memory means that the whole memory is divided into n number of memory units

2 – Way Memory Interleaving

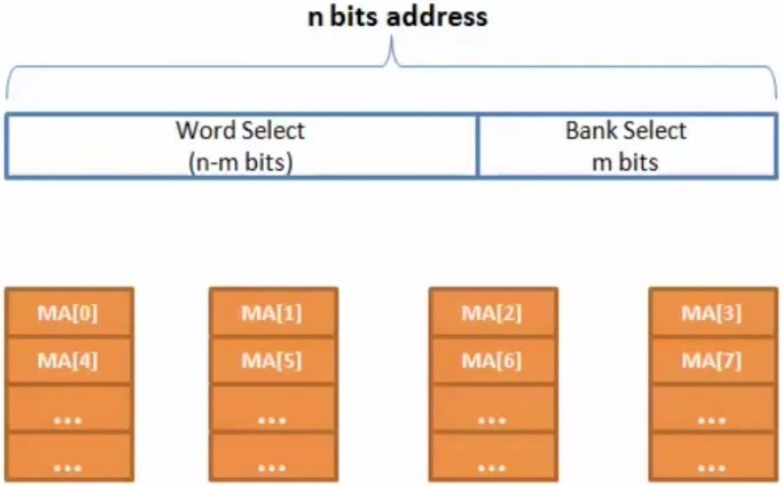


4 – Way Memory Interleaving

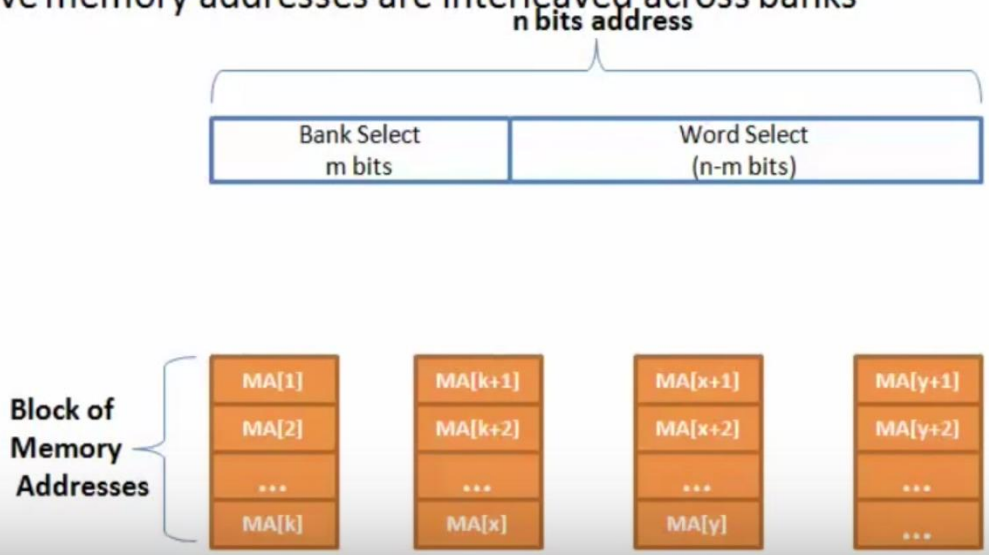


Types of Memory Interleaving

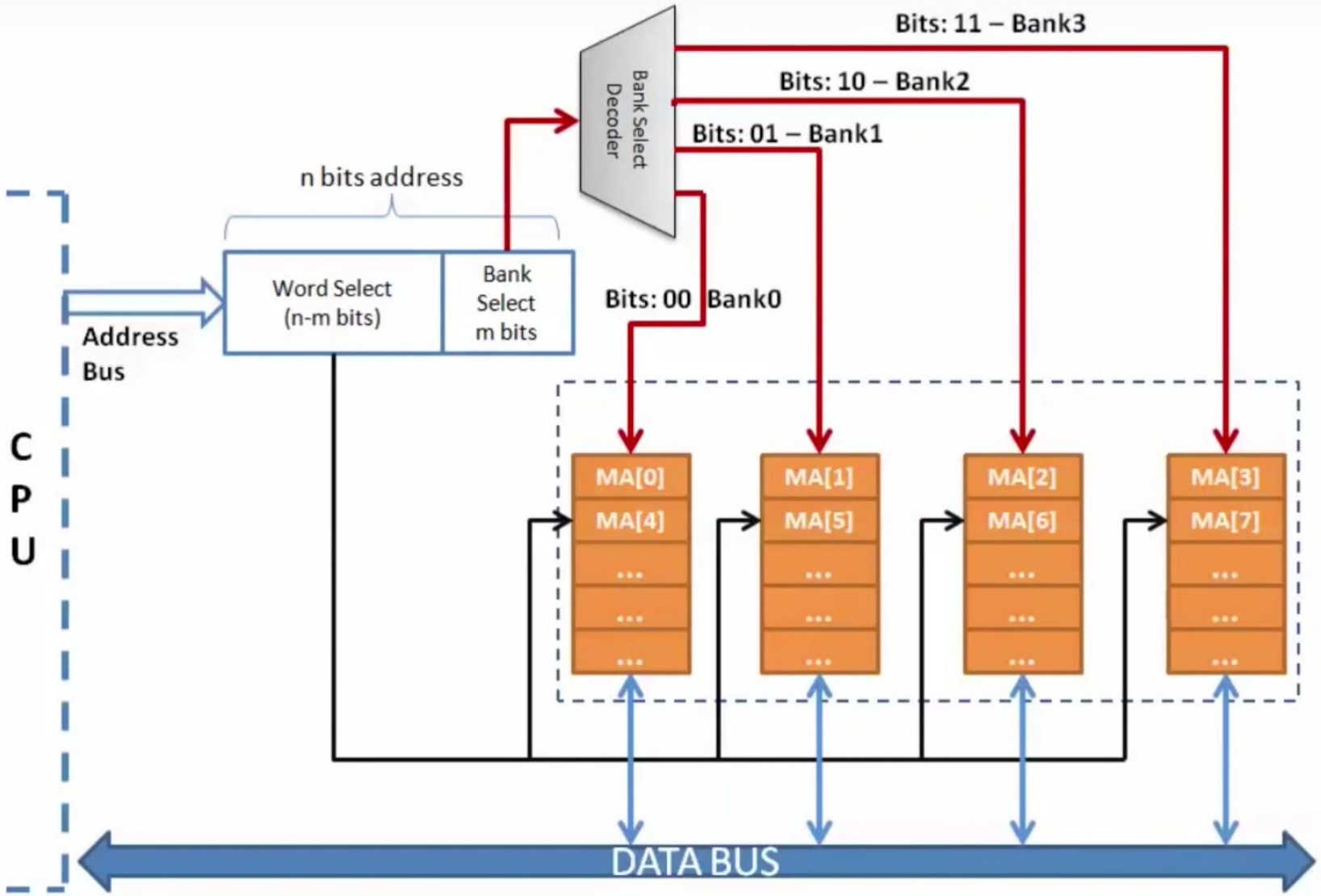
1. **Low Order Interleaving** : Least significant address bits are used for bank select. All successive memory addresses are interleaved across banks



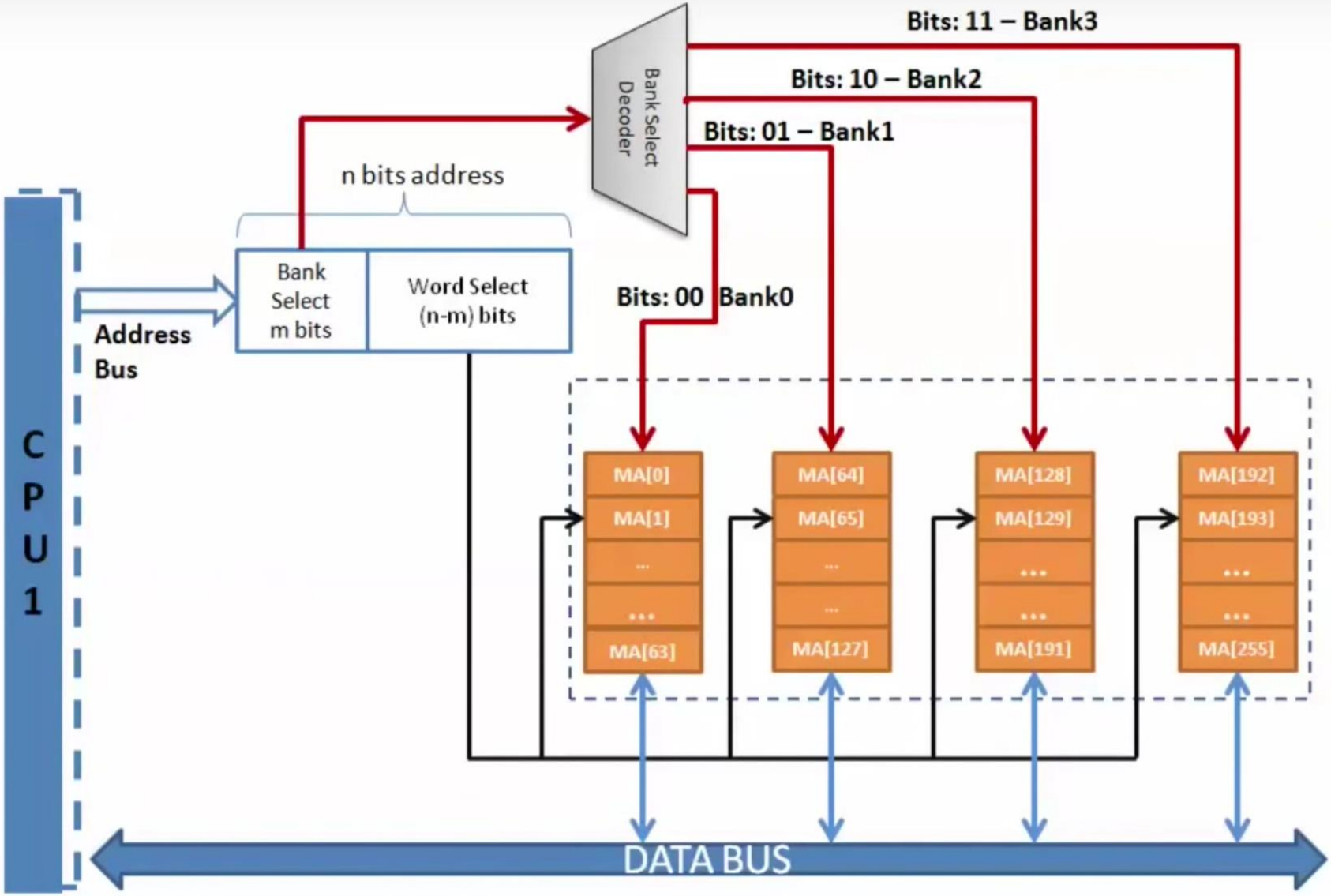
2. **High Order Interleaving** : Most significant address bits are used for bank select. Block of successive memory addresses are interleaved across banks



Low Order Interleaving



High Order Interleaving



- *Low-Order interleaving supports pipelined block access of contiguous memory locations*

- *Under low-order interleaving technique graceful degradation of performance is achieved*

Example of Low-order interleaving – Fault-tolerance:
8 memory modules viewed as banks

8-way implies 8 banks – 1 module per bank – failure of 1 module can provide 7 words;

4-way implies 4 banks – 2 modules per bank – failure of 1 module can provide 6 words;

2-way implies 2 banks – 4 modules per bank - failure of 1 module means 1 bank failure; but this can still provide 4 words;

1-way implies 8 modules per bank – failure of 1 module will be a total failure of the system (total of 1 bank)

- *High-Order interleaving* is preferred for shared memory systems. *With supporting hardware circuitry, when one CPU is accessing a memory module other CPU can access other memory module, especially under read modes.*

- *High-order interleaving technique does not support pipelined block access of contiguous locations;*

Stay tuned for Part 2 …