Chapter 4. Design Principles of Bus, Cache, and Shared Memory Organizations

Contents of this chapter

- Bus timing and control operations
- Addressing and timing protocols
- Bus arbitration policies
- Cache memory organization
- Techniques of mapping onto the cache

- Replacement technique – LRU, LFU
- Virtual memory - concept and use of PTs and TLBS
- Shared memory organization
- Memory allocation schemes

Reference : Kai Hwang's book, Chapter 5

Pages: *Reading assignment: Page 215 – Functional modules; Page 221 – Transaction mode; page 221(sect 5.1.4) o 224; Use the slides for the cache memory*

and virtual memory related stuff; 238-240;

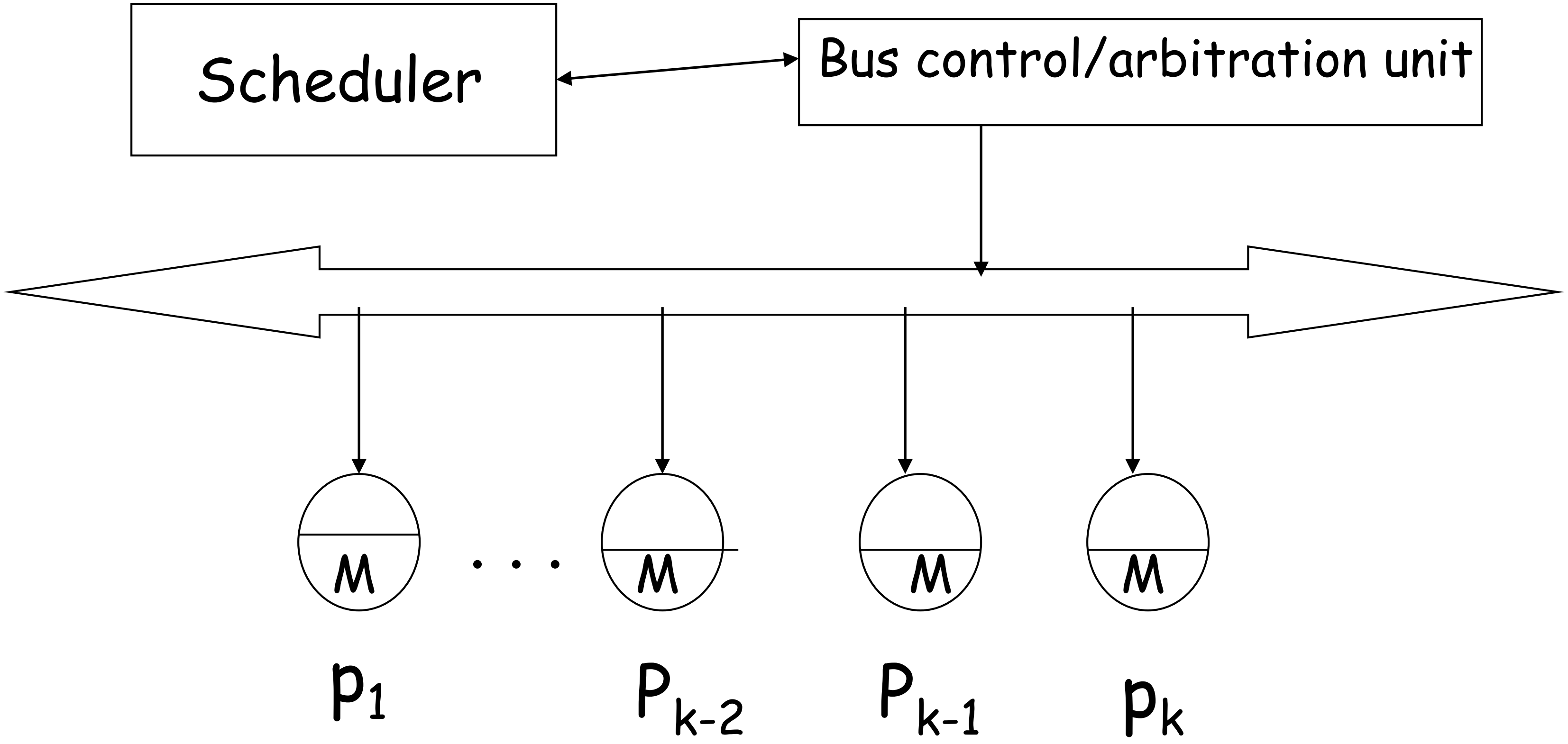
Fig 5.16 (a) (page 241); Example 5.7(pg: 242)

- *Shared memory systems - use the slides flashed*

Backplane Bus Systems

- Bus based system is a shared environment.
- Available bandwidth is shared among the nodes hooked onto the bus. This means that the *effective bandwidth* available to each processor is *inversely proportional* to the # of nodes in the system.

Obviously, this means that a bus system has to be small in size. 4 to 16 processors seem to be a modest size based on today's network technology.



General purpose bus system

Backplane bus specification

- Interconnects the nodes, storage and peripheral devices in a *tightly coupled h/w configuration*

- Without disturbing any internal activities of a node, the devices must be able communicate

- Timing protocols must be established to resolve any possible contention and to have a fair arbitration among the nodes

- All possible signal lines must be considered

Refer to Fig. 5.1 on page 214

Data Transfer Bus (DTB):

- data, address, and control lines form this set in a bus.

- The addressing lines are used to broadcast the data and the device addresses.

- The number of address lines is proportional to the log of the size of the address space.

- Different addressing modes possible

- The # of data lines are proportional to the memory word length.

Example : IEEE Bus Specifications 32/64 address lines, 32/64 data lines; muxing possible between data and address transfer; control lines are used to indicate read/write, timing control, and bus error conditions

Bus Arbitration and Control:
Process of assigning control of DTB to a node that requests is called *arbitration*.

To co-ordinate the arbitration process, we have dedicated lines

Interrupt is possible and these are entertained on separate lines. These are prioritized.

Synchronization is also a part of this activity and this is handled by dedicated lines

Utility lines provide signals that help to co-ordinate the power-up and power-down sequence of events.

Bus Arbitration Mechanisms

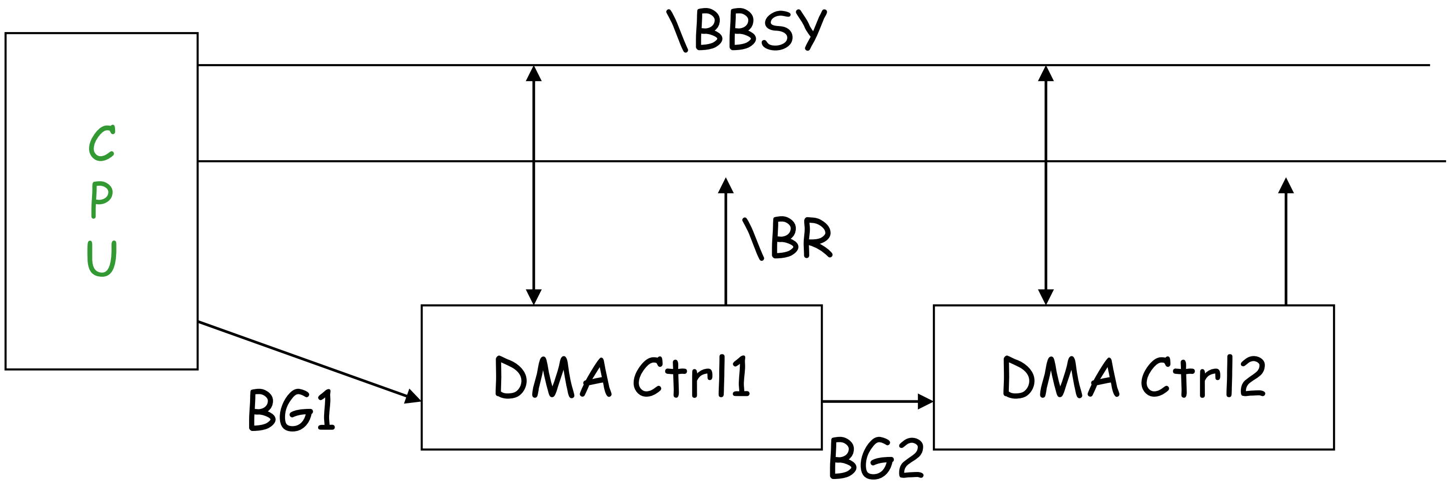
Bus arbitration is a process by which contention due to the use of bus by more than one device is prevented. *For instance, a DMA transfer uses the same address, data, and control lines used by the processor to communicate with the memory and the I/O device.*

Therefore we need to have a means to coordinate the use of these lines, to prevent 2 devices from initiating transfers at the same time.

A device that is allowed to initiate data transfers on the bus is called the *master.* Clearly, only one master can exists at a time.

When the master relinquishes the control, the other devices can acquire this status. The bus arbiter performs the required scheduling function.

Example: DMA transfer arbitration



The above figure indicates a basic arrangement when the CPU contains the bus arbiter circuitry.

Normally the processor is the bus master, unless it grants the ownership to any of the devices.

- To become a master(say a DMA controller) : Activate the bus request line \BR (read as 'BR bar'). The signal on the BR line is a logical OR of the requests from all the devices connected to it.

- When BR line is activated, the processor activates the Bus-Grant signal, BG1, indicating the DMA controllers that they may use the bus as soon as it becomes free. This signal is received by all the devices via a daisy-chain arrangement.

- Thus, if DMA controller 1 is requesting the bus, it blocks the propagation of this grant signal to its successor. Otherwise, passes the signal downstream.

- The current bus master indicates to all the devices that it is using the bus by activating the \BBSY signal.

- Thus, after receiving the bus grant signal by a device, it waits the \BBSY line to be inactive, for the bus to become free. After this, the device again activates the \BBSY line to prevent other devices from using the bus.

- Since the role of the arbiter is to ensure contention-free scheme, several policies are possible:

Fixed-priority scheme: If there are 4 bus request lines, BR1 through BR4, then BR1 is given the top priority.

Alternatively, a rotating priority scheme is possible to give all the devices an equal chance. Rotating priority means that, after a request on line BR1 is granted, the priority order becomes 2,3,4,1 for the bus request lines.

All these arbitration schemes are referred to as *centralized* arbitration schemes.

Distributed arbitration schemes do exists. A common method distributed scheme is as follows.

Each device is assigned a 4-bit number, as an ID. When one or more devices contend for the bus, they assert the *Start_Arbitration* signal and place their 4-bit ID on the four request lines.

A winner is selected by means of a combinatorial logic and every device knows who is the current master at the end of this "election" phase.

As an example, consider two devices A, B contending for the bus with IDs, 5 and 6, respectively.

Device A transmits the pattern 0101 and the device B transmits 0110, and hence the net pattern seen on the bus lines as a result of interaction of these signals is 0111.

When this pattern stabilizes on the bus lines, each device compares its ID with that of 0111 starting from the MSB. If it detects a change in any bit position, it disables its drivers and comes out of election process.

Thus, in our case, device A sees a change in bit position 1 and hence, the resulting pattern on the bus effectively becomes 0110, the address of B. Thus, B wins the election and becomes the master.

Note that this is one particular scheme, and there could be schemes that allow random address generation and this may be fair in some situations. Otherwise, in the above policy, whenever A and B place requests, B always wins, however important the job of A is!

Functional Modules: [Refer to your Book - Page: 215 – *reading assignment*]

Timing in Bus Transactions: Synchronous and Asynchronous

A typical bus consists of data, address, and control lines. The control signals involved in data transfers specify two types of information

- nature of the transfer
- timing

The first of these is referred to as mode of transfer involves the specification of whether a Read or a Write operation is to be performed.

The second component carries the timing information. These signals specify the times at which the processor and the I/O devices may place data on the bus or receive from the bus.

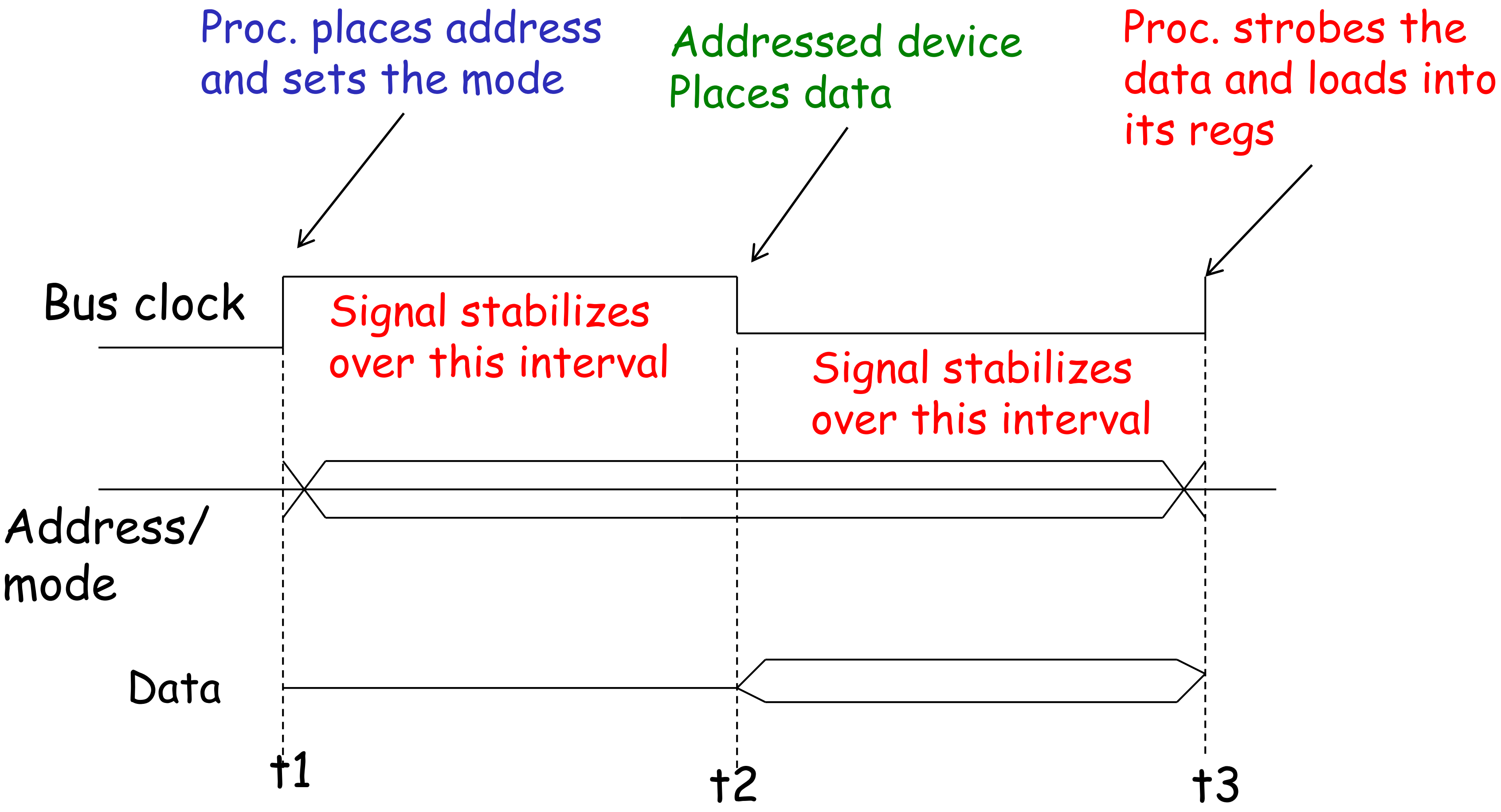
These can be broadly classified as

- Synchronous
- Asynchronous

Synchronous bus: All devices derive the timing information from a common clock line. Equally spaced pulses on this line define equal time intervals; each interval constitutes a bus cycle.

During each bus cycle, one data transfer can take place.

For a read operation, consider the timing diagram shown below.


Proc. places address and sets the mode
Addressed device Places data
Proc. strobes the data and loads into its regs
Bus clock
Signal stabilizes over this interval
Signal stabilizes over this interval
Address/ mode
Data
t1
t2
t3

At time t1, the processor places the device address on the address lines and sets the mode of operation (read/write). This information travels over the bus at a speed determined by the electrical characteristics of the system.

The clock pulse width (t2-t1) should be carefully chosen such that it is greater than the maximum propagation delay time between the processor and any of the devices connected to the bus.
It should also be wide enough to allow all the devices to decode the address and control signals so that the addressed device can be made ready to respond at time t2.

The addressed device recognizing that an input operation is requested, places its input data on the data lines at time t2. At the end of the clock cycle, at t3, the processor strobes the input data from the data lines and loads into its intended input buffer/register.

For the data to be loaded correctly into any storage device(in our case the register), the data must be made available for a period greater than the set-up time delay of the device. Hence, the period (t3-t2) must be greater than the maximum propagation time on the bus plus the set-up time of the input register of the processor.

Asynchronous bus transfer:

An alternative scheme is by *handshaking mechanism* which is employed in the case of asynchronous transfer mode.

Common clock is eliminated in this case. The clock line is replaced by two control lines - Ready and Accept lines.

In principle, the data transfer controlled by a handshake protocol is as follows.

- The processor places the address and the mode of operation on the bus.

- Then, it indicates all the devices that it has done so by activating the Ready line.

- When the addressed device receives the Ready signal, it performs the required operation and then informs the processor it has done so by activating the Accept line.

- The processor waits for the Accept signal before it removes its signals from the bus. In the case of a read operation, it also strobes the data into its input buffer.

Try this exercise ! Draw the timing diagram describing the above protocol for a read operation. Explain the events happening at various time instants within a bus cycle.

Synchronous and Asynchronous Timing:

Refer to Fig. 5.3 on page 218

Bus Arbitration, Transaction, and Interrupt:

Refer to Fig. 5.4 on page 219

Independent Request handling:

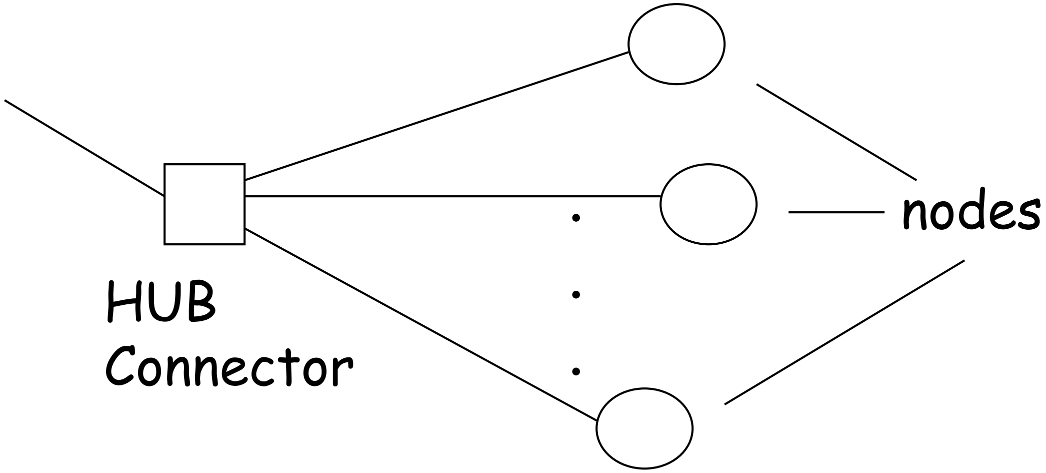
Refer to Fig. 5.5(a) on page 220

Note: We have already seen these.

IEEE Futurebus+ Standards

Reading assignment : Page 221-224

A HUB is also referred to as a bus, in practice.



Memory Systems Design Principles

Basic Stuff here!!!!

Note: Slides 31 to 42 - ***For you to read some very basic stuff…***

Some basic concepts and facts !

- Max size of the memory that can be used in any computer is determined by the addressing scheme.

  A 16 bit computer generates 16-bit addresses and is capable of spanning up to 2^{16} = 64K memory locations.

- Modern computers are byte addressable.

Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11

Organization of the main memory in a 32-bit byte-addressable computer

Big-endian arrangement

- The main memory (MM) is usually designed to store and retrieve data in word-lengths

Note: The number of bits actually stored and retrieved in one MM access is the most common definition of the word length of a computer.

Example

Consider a byte-addressable computer with the structure shown in the above figure. When a 32-bit address is generated from the CPU and sent to MMU, the higher order 30 bits determine which word to access. If a byte quantity is specified, the low order 2 bits of the address specify which byte location to access.

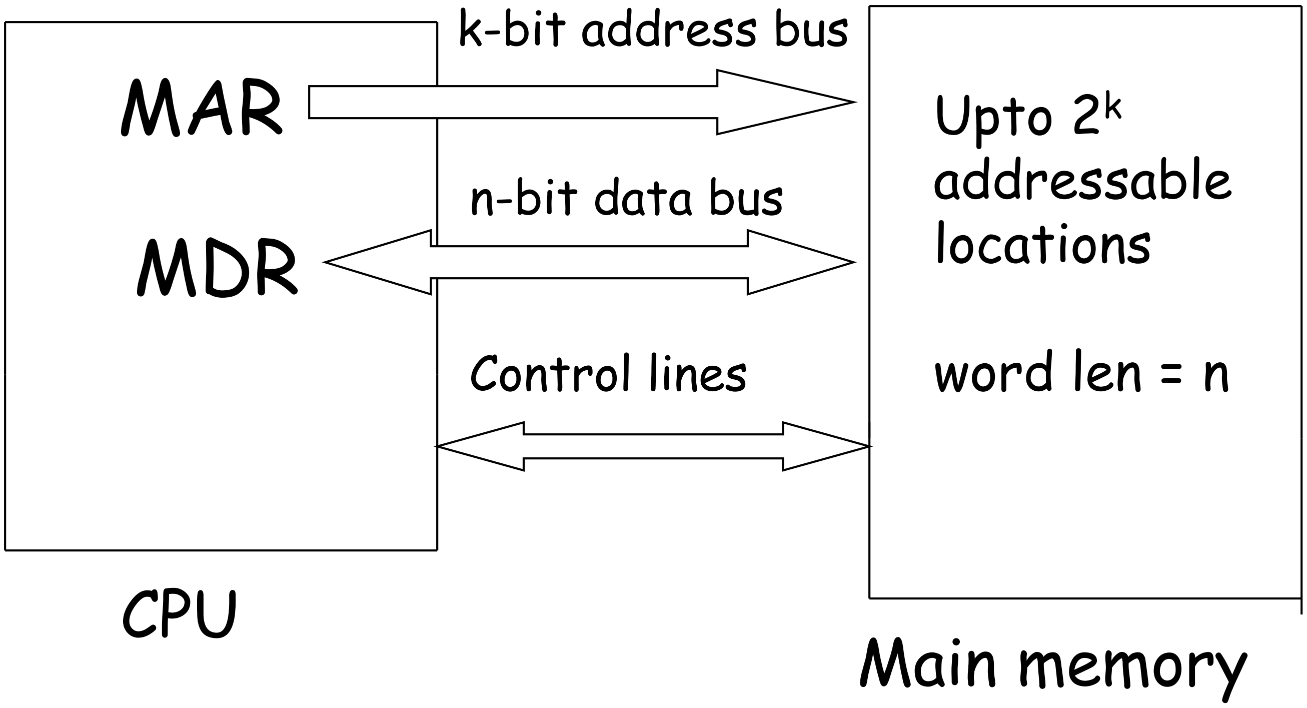
- Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (memory address register) and MDR (memory data register).

- If MAR is k bits long and MDR is n bits long, then the memory unit may contain up to 2^k addressable locations.

- During a memory cycle, n bits are transferred between memory and CPU. This transfer takes place over processor bus which has k address lines and n data lines.

- The processor bus also includes control lines Read, Write, and Memory function complete(MFC) for co-ordinating data transfers.

- In a byte addressable computers, another line may be present to indicate that the transferred word is just a byte rather than a full word of n bits.



CPU-Memory interaction:

1. CPU initiates a mem operation by loading the appropriate data into registers MDR and MAR.

2. It then sets either read/write memory control line to 1.

3. When the required mem operation is completed the mem control circuitry indicates this to the CPU by setting MFC to 1.

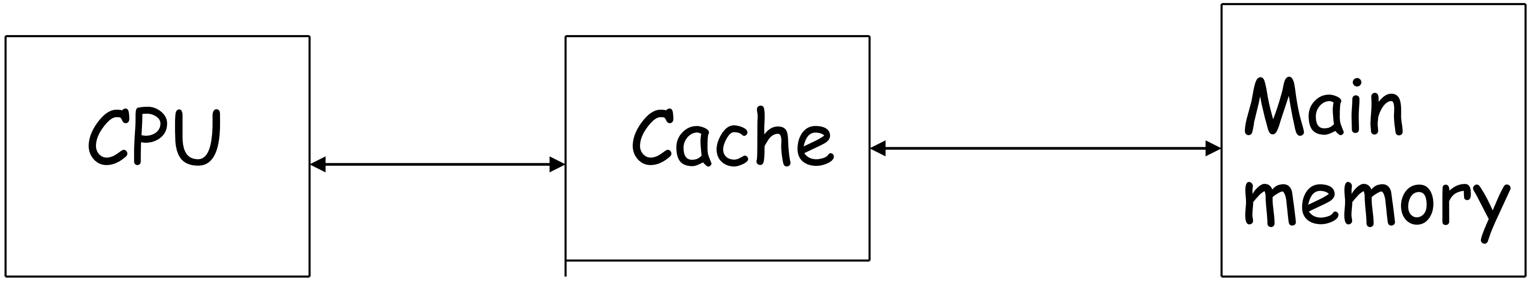
Some common terminology

A useful measure of the speed of the memory units is the time that elapses between the initiation of an operation and the completion of that operation, for example, time between the Read and MFC signals. This is referred to as *memory access time*.

Another useful measure is the *memory cycle time*, which is the minimum delay required between the initiation of two successive memory operations, for example, between two successive Read operations.

Cycle time is usually > access time, depending on the implementation details of the memory units.

Cache memories



Why cache?

- CPU processes the instr. and data at a much faster rate than they can be fetched from mem unit.

- Then, memory cycle time is the bottleneck of the system.

Note that even if we prefetch and keep some of the instructions and data into a location, unless the memory cycle time is reduced, the problem cannot be solved.

Cache memory helps to solve this problem. This is a small piece(!!) of memory placed between CPU and the MM unit so that CPU always checks for the required data/instr. in cache first and then in the MM. This is a very fast memory unit, specially designed to match the speed of the CPU.

- Effectiveness of the cache mechanism is based on a property of computer programs called the *locality of reference.*

Typical analysis of programs revealed a nice property that maximum time is spent on instructions that are repeated frequently .

This property manifests in two ways - <u>spatial and temporal</u>.

The spatial property means that a recently executed instruction is likely to be executed again very soon, and instructions in “close” proximity are also likely to be executed soon.

The temporal property suggests that whenever an information item is first needed, this item should be brought into the cache where it will *hopefully* remain until it is needed again.

Spatial aspect suggests that instead of bringing one instruction, bring a set of instructions “close” to the currently needed one.

We will use the term *block* to refer to a set of contiguous addresses of some size.

Note : Some authors use *cache line* to refer to a cache block, in the literature.

General working principle of cache based systems

Refer to the figure on Slide 40. 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 locations from this block, the desired contents are read directly from 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.

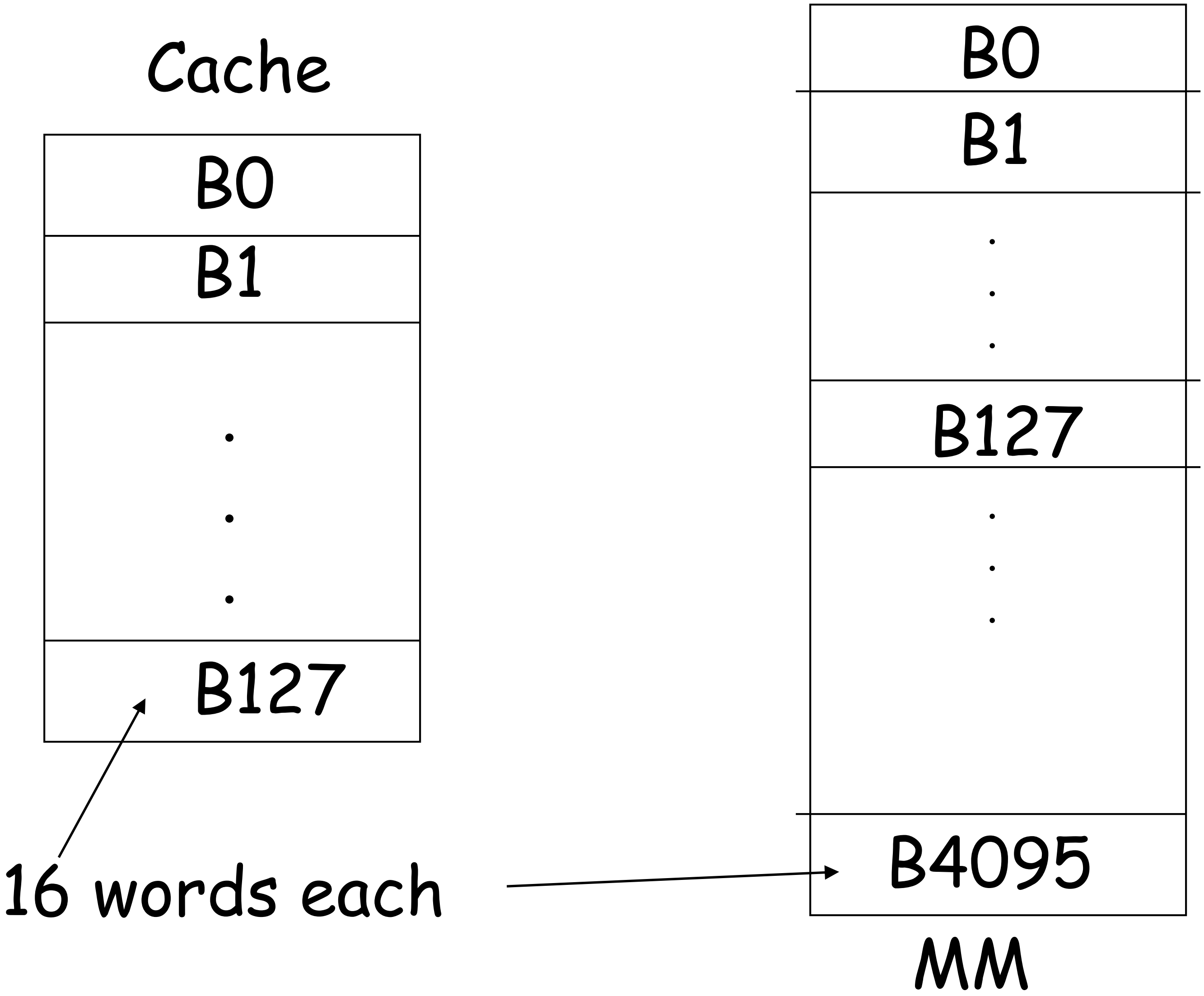
Let the cache consists of 128 blocks of 16 words each, for a total of 2048 (2K) words.

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,

5	7	4
Tag	Block	Word

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

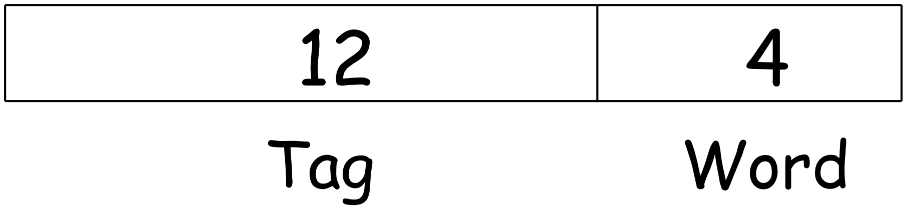
Note that the tag field in the above example is nothing but the higher order 5 bits of the word address .

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.

(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,

Tag	Set	word
6	6	4

MM address

Memory Mapping Functions (Cont'd)…

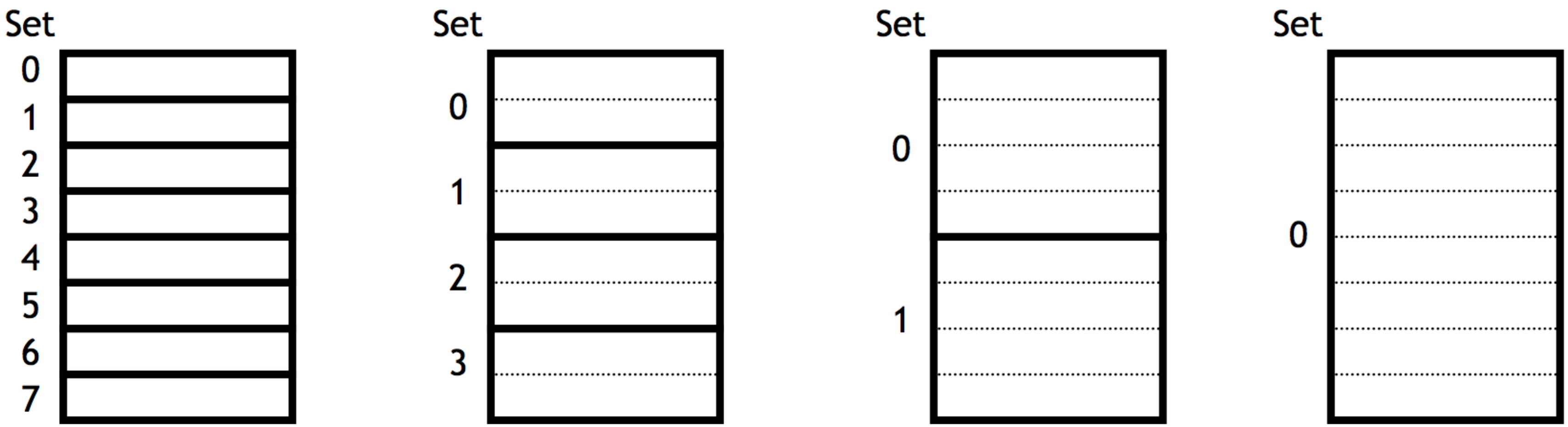
From Direct Mapped to Fully Associative

1-way
8 sets,
1 block each

2-way
4 sets,
2 blocks each

4-way
2 sets,
4 blocks each

8-way
1 set,
8 blocks



direct mapped

fully associative

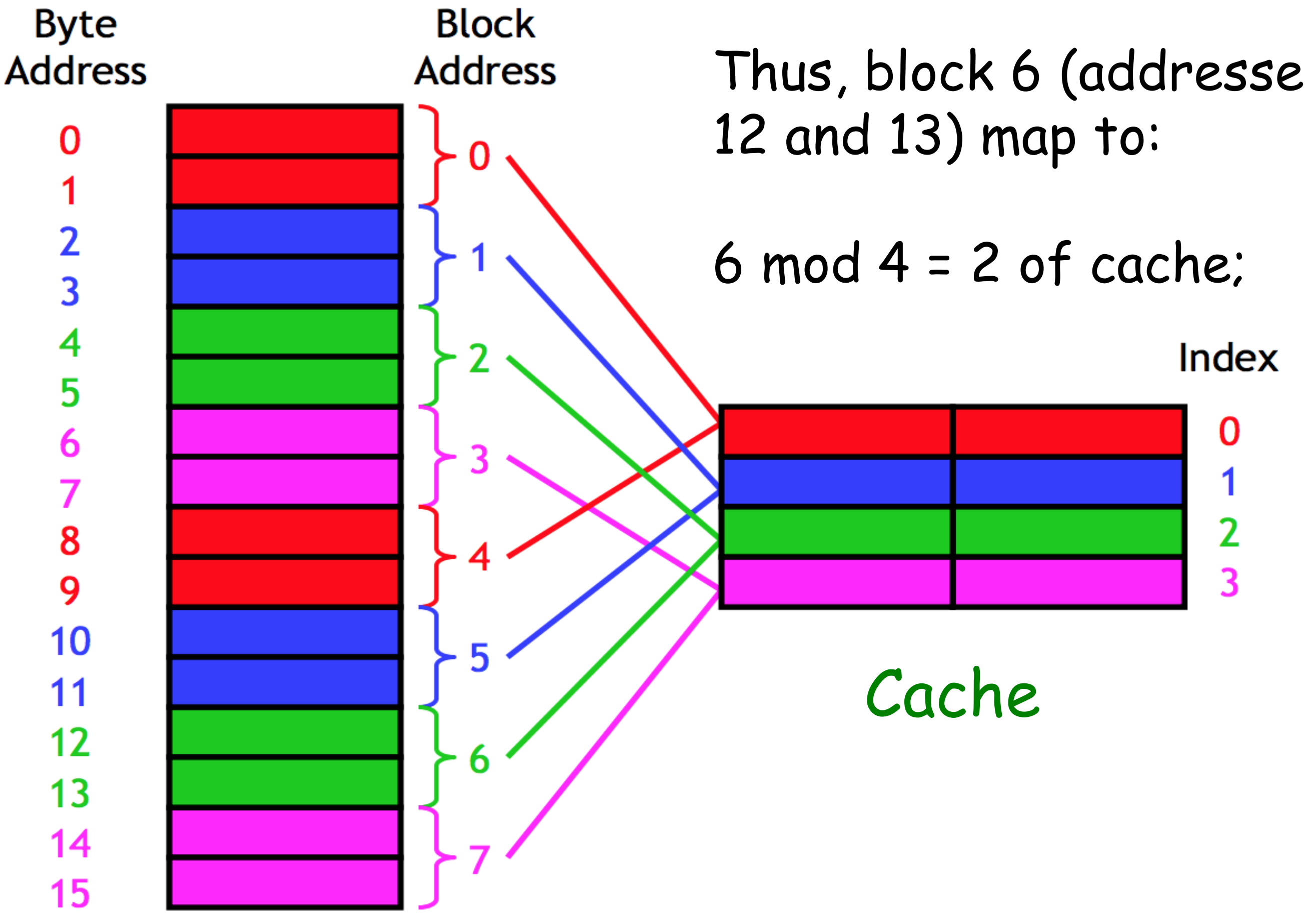
Direct mapped cache – Identifying Valid & Offset bits

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;



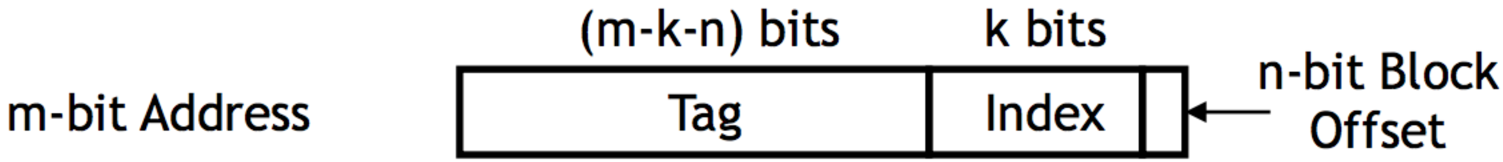
Direct mapped cache… (cont’d)

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:

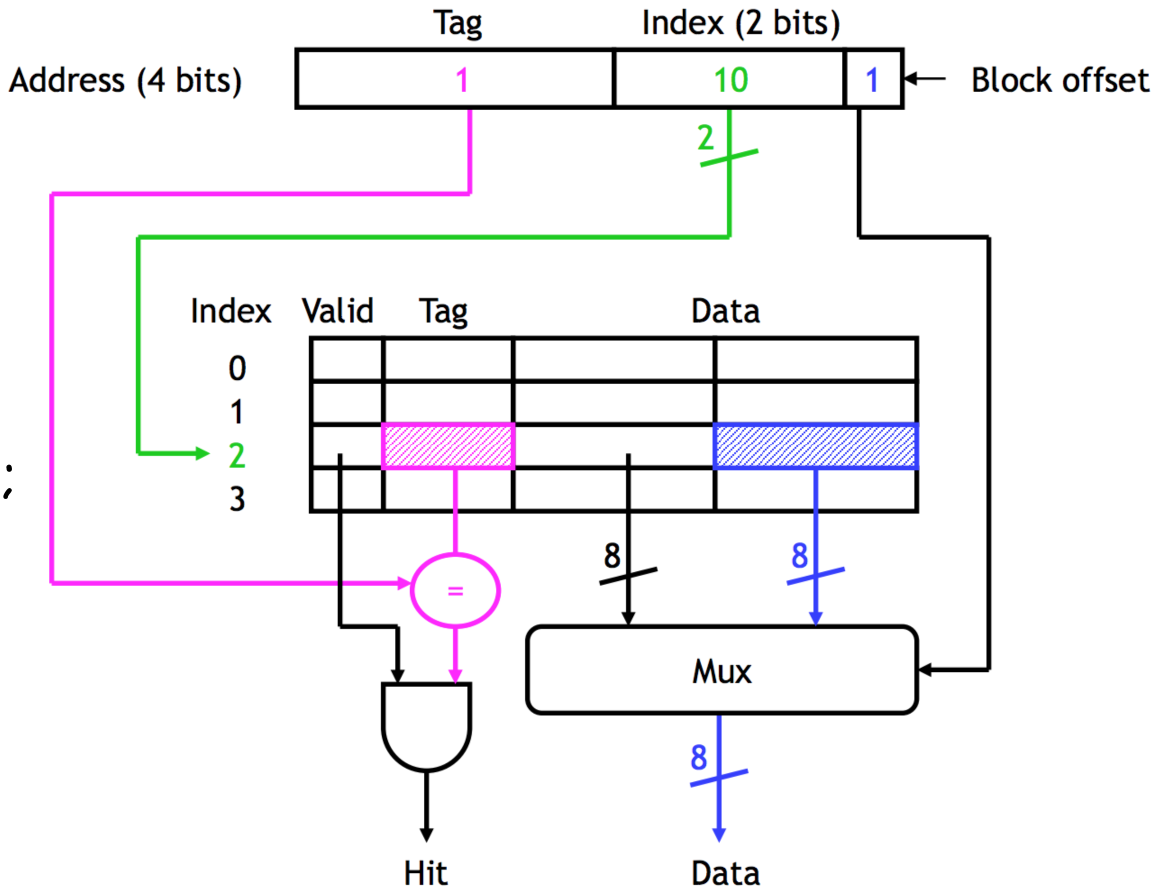


Direct mapped cache… (cont'd)

4 bit addr

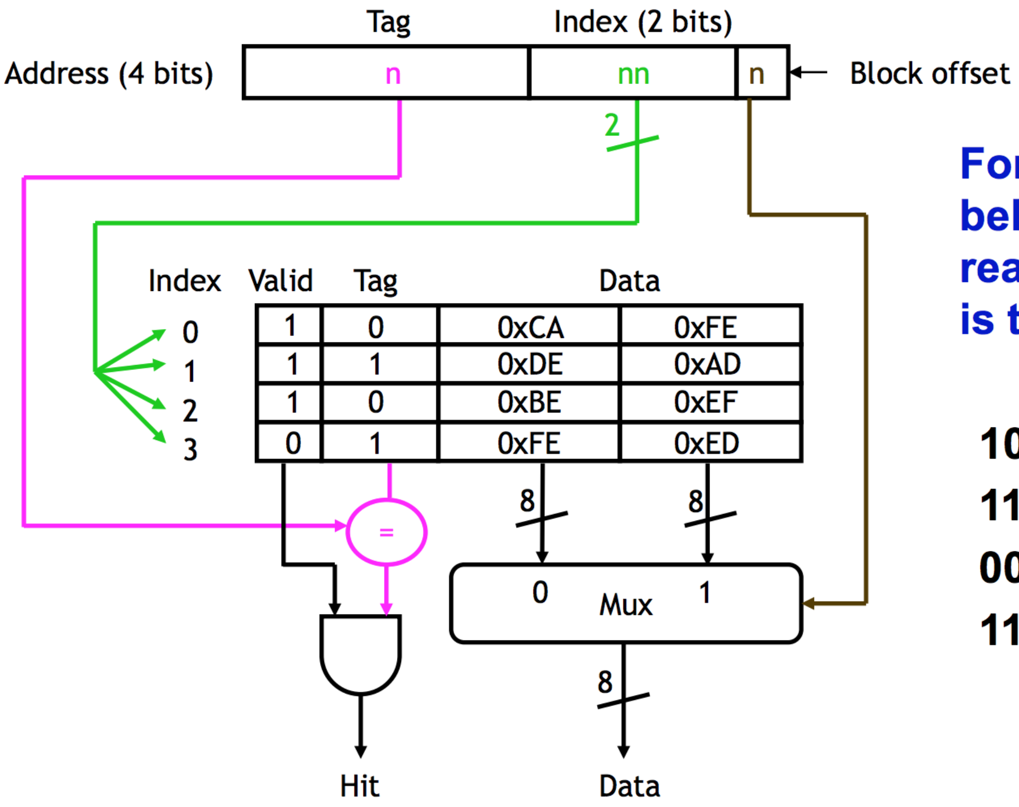
Block 6 comprising Addresses 12 & 13 Map to Block 2 (10) of cache;

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!



Index	Valid	Tag	Data	
0	1	0	0xCA	0xFE
1	1	1	0xDE	0xAD
2	1	0	0xBE	0xEF
3	0	1	0xFE	0xED

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

1010

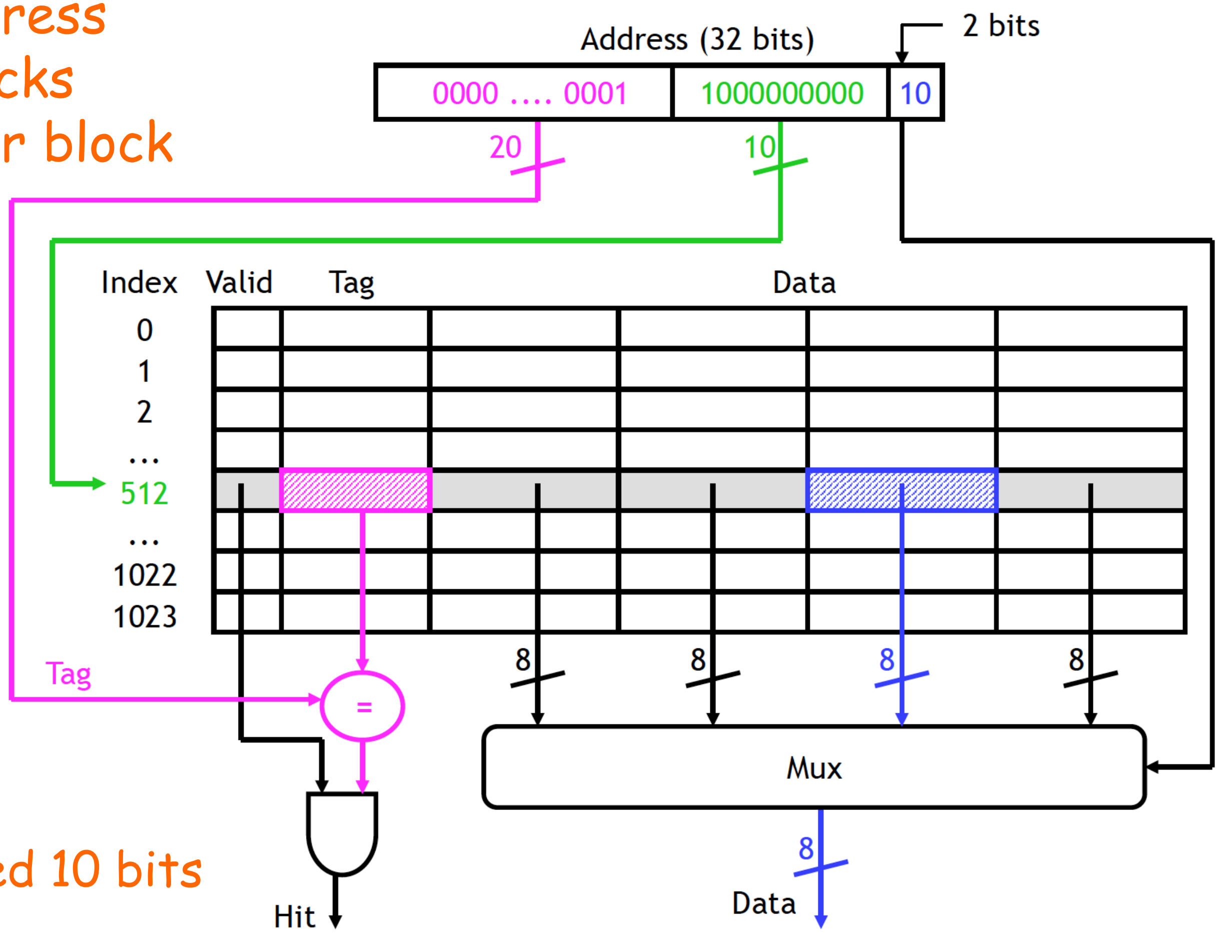
1110

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

time

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

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

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

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, with the new one;

Consider the same given Seq; Determine the LFU pages;

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

4 faults

Page fault;
Replace 7 / 1 / 2
as all of them
are LFUs
Assume 7 is
replaced

Page fault;
Replace 3 / 1 / 2
as all of them
are LFUs
Assume 2 is
replaced

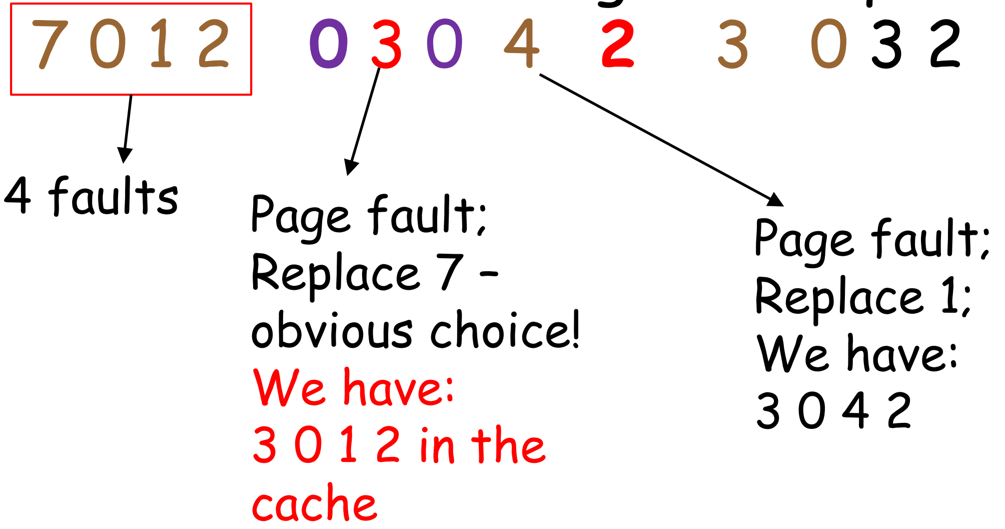
Continue replacing LFUs
as earlier;

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.

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 the total number of faults.

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 following:

At time 6, the set of pages in the 3-frame memory is not a subset of the set in the 4-frame memory. This means the 4-frame memory will producing a page fault that does not occur in the 3-frame memory. You will again see this phenomena at t=7 and t=10.

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.*

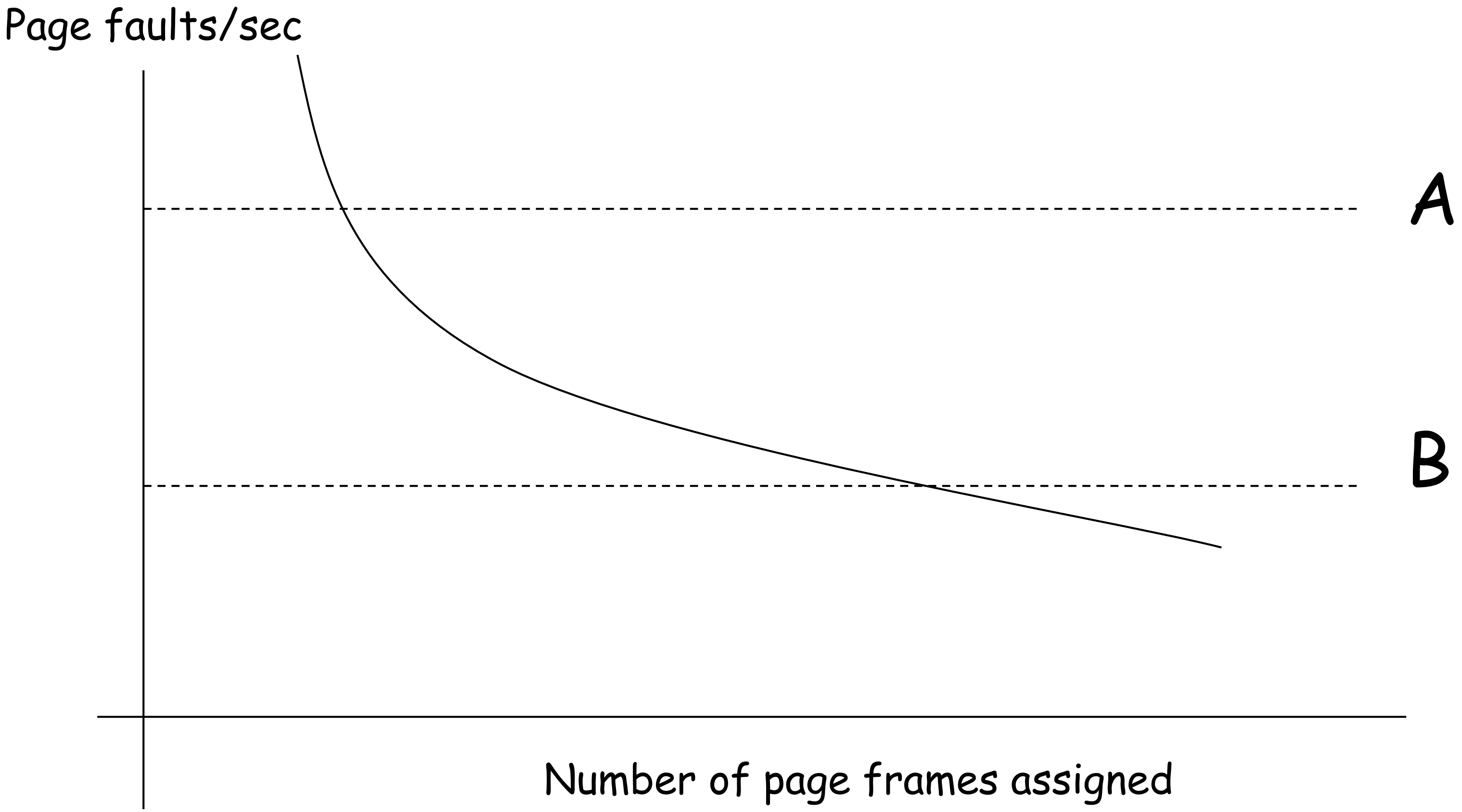
Equal sharing strategy: In this approach, the OS periodically determines the number of running processes and allocate each process an equal share.

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.

Effect of page size

Page size is often a parameter that can be chosen by the OS designers.

Even if the hardware has been designed with, for example, 512-byte pages, the OS can easily consider pages 0 and 1, 2 and 3, 4 and 5, and so on, as 1K pages by always allocating two consecutive 512-byte page frames for them.

Determining the optimum page size requires a careful balance between several competing factors.

For instance, a randomly chosen text, data, will not fill an integral number of pages. On an average, half of the final page will be empty. The extra space in that page is usually wasted (*internal fragmentation*). Thus, with n segments in memory and a page size of p bytes, np/2 bytes will be *wasted* on internal fragmentation.

Virtual Memory Technology

Though the modern day computers have enough MM, still the amount of applications that run demand a large working space.

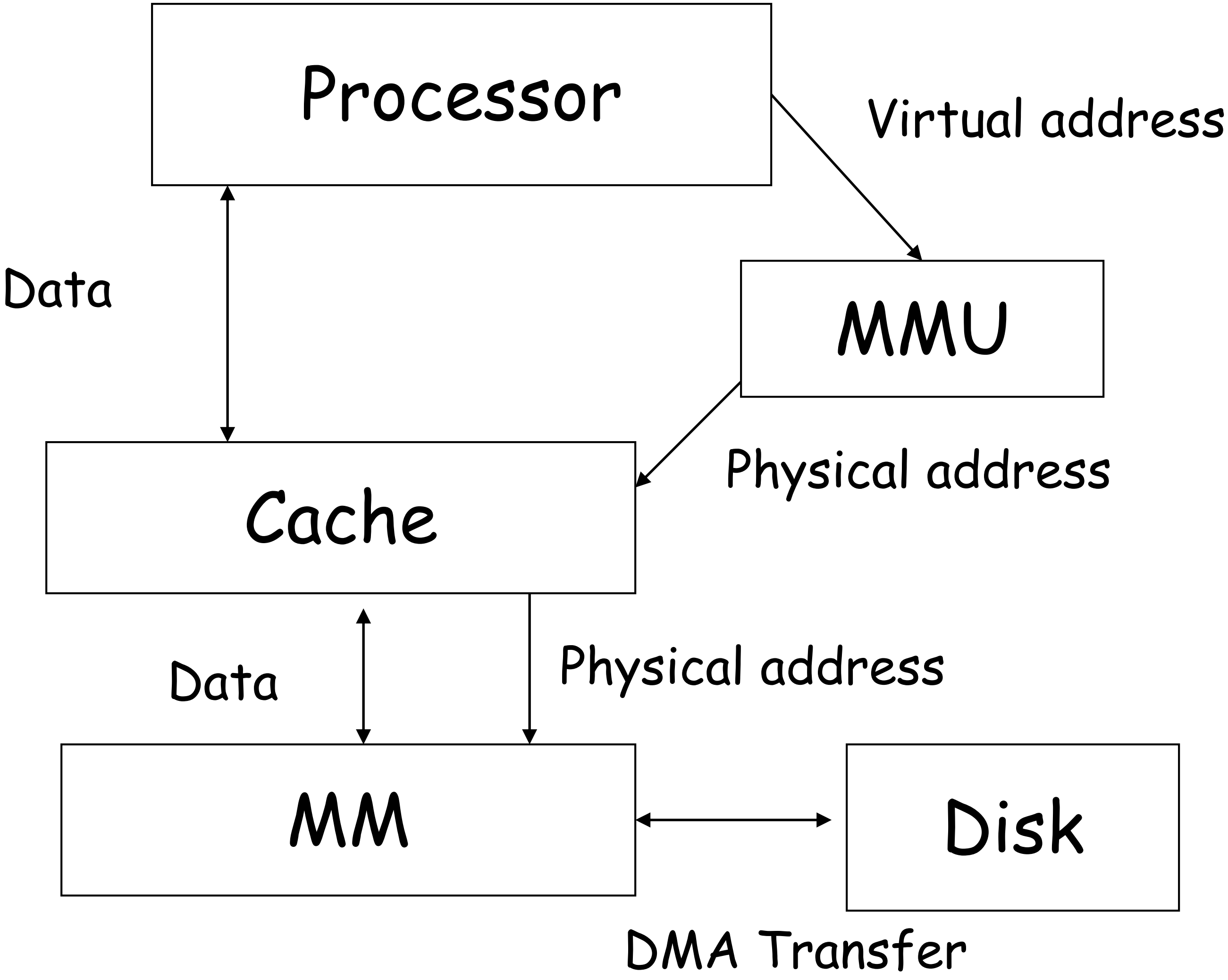
Also, usually, the physical MM is not as large as the address space spanned by an address issued by the processor.

When a program is to be executed, it has to be brought into the MM . The OS moves the data/pgm between the MM and the secondary storage devices

Techniques that move the pgm or the data automatically to the MM when they are required by the CPU are called *virtual memory techniques.*

The binary addresses that the processor issues for either instructions or data are called *virtual or logical addresses.*

These virtual addresses are *translated* into physical addresses by a *combination of h/w and s/w components*.



Processor

Virtual address

Data

MMU

Physical address

Cache

Data

Physical address

MM

Disk

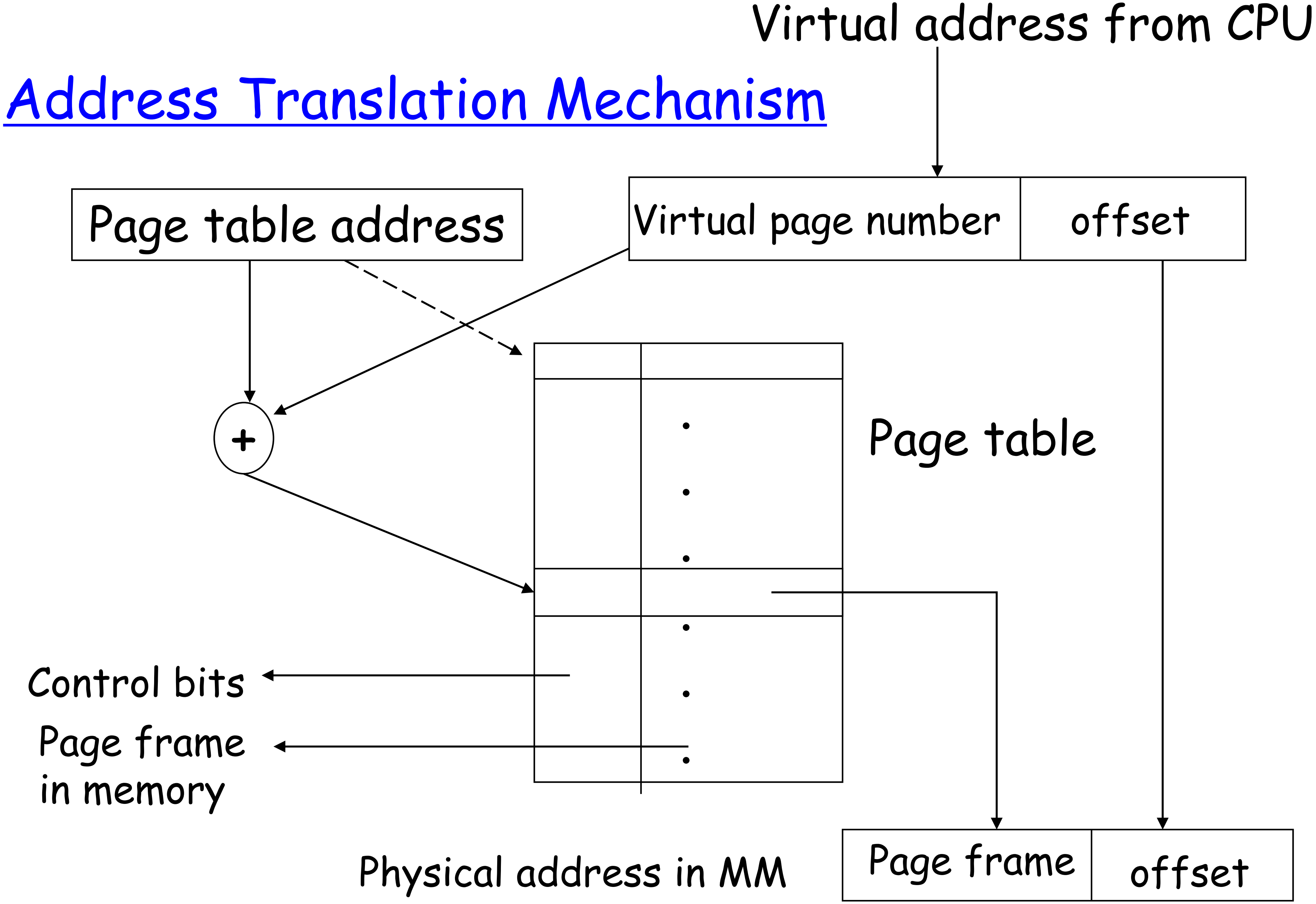
DMA Transfer

Note:

- A cache bridges the speed gap between CPU and MM and is implemented in h/w.

- The VM mechanism bridges the speed and size gap between MM and secondary storage and is implemented using s/w.

Address Translation Mechanism



Virtual address from CPU

Page table address

Virtual page number

offset

+

Page table

Control bits

Page frame in memory

Physical address in MM

Page frame

offset

Estimating the size of the Page Table

Each process uses different number of pages.

This means, memory space demands of each process is different.

Question: For every page that belongs to a process we need to have an entry in the PT. *When each process uses different number of pages, how do I estimate the size of a page table*?

P: Size of a page in bytes;
S: size of a process in bytes;
e: #of information bits needed per page in the PT (in bytes);

S/P - # of pages used by the process;
Se/P: # of bits;
P/2: Overhead due to internal fragmentation

Note: On an average P/2 amount of memory is wasted in the last page in a process and this is referred to as an internal fragmentation;

Estimating the size of the Page Table

Thus, the total overhead is: $Se/P + P/2$

1st term has an inverse relationship in P;
2nd term is directly proportional to P;

This prompts us that an optimal value must be somewhere In between!

Determine an optimal P*:

This yields:

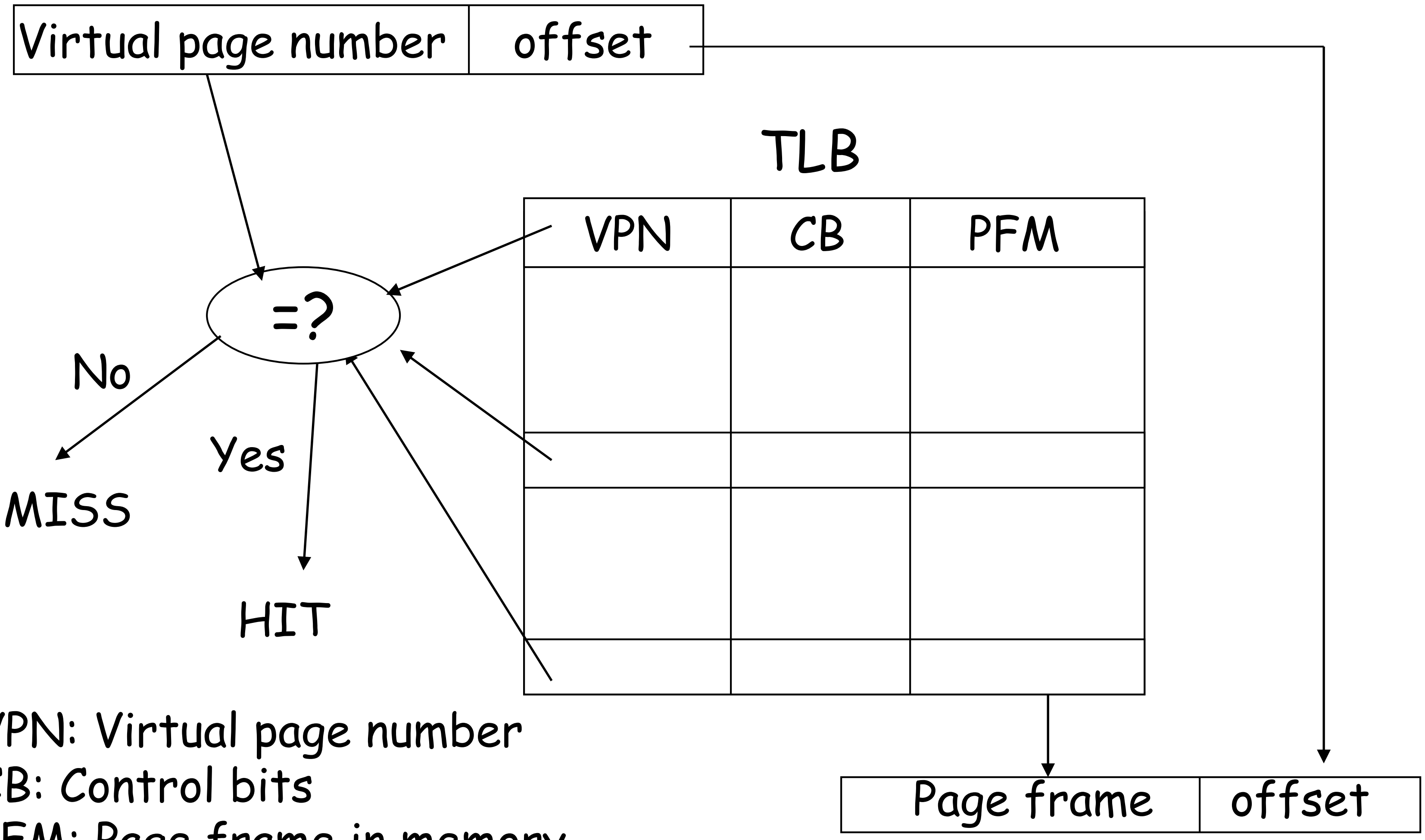
$$-Se/P^2 + \frac{1}{2} = 0;$$

implies

$$P^* = (2.S.e)^{1/2}$$

As an example, S = 128K; e = 8 bits, implies: $P = (2.S.e)^{1/2} = 1448$ bytes! As a thumb rule, in practice, a 1K or 2K size will be used depending on other factors such as disk speeds & other overheads;

Use of Translation Look-aside Buffer (TLB)



VPN: Virtual page number
CB: Control bits
PFM: Page frame in memory

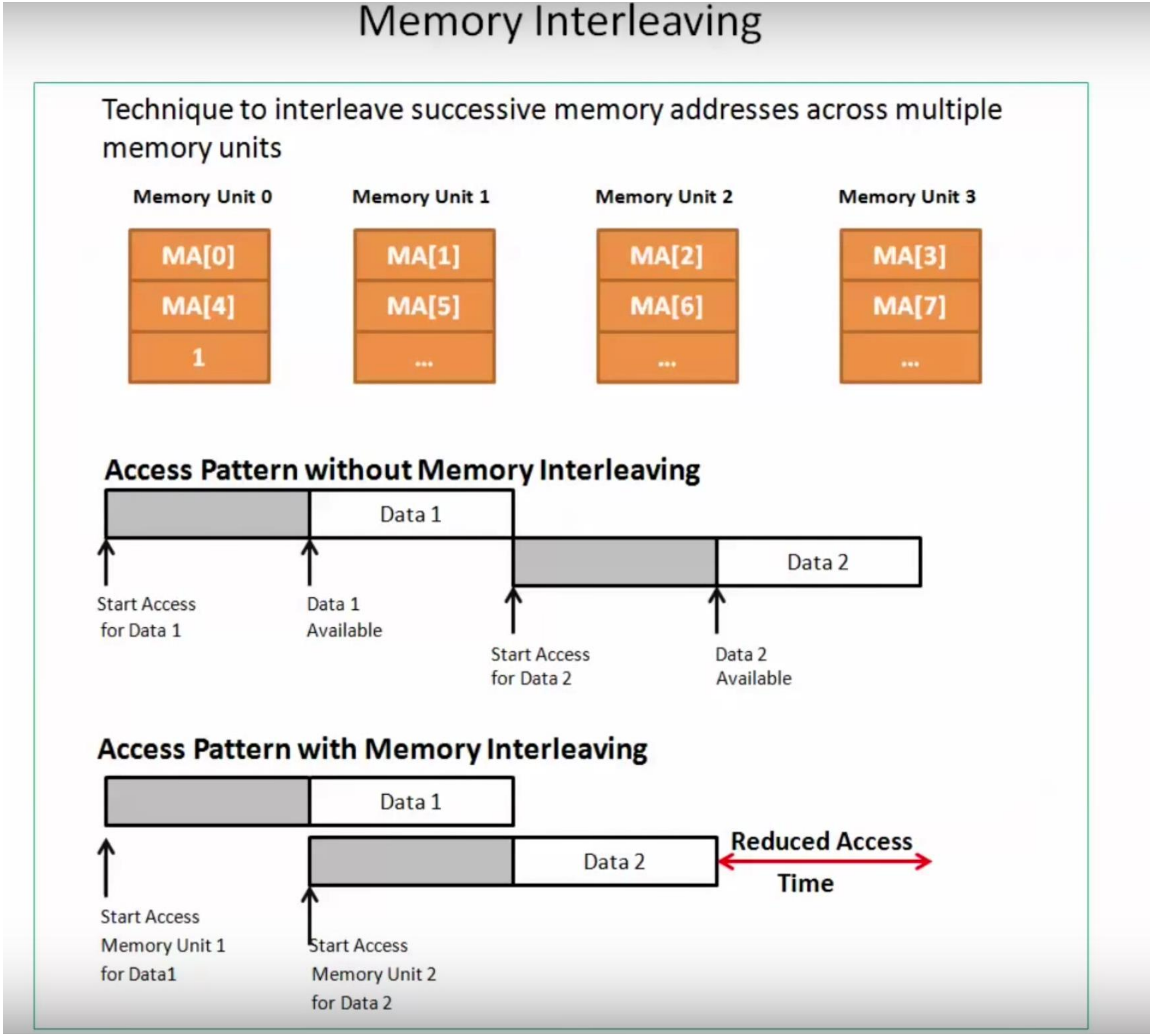
For detailed notes on address translation and TLB, download the notes from our course site

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

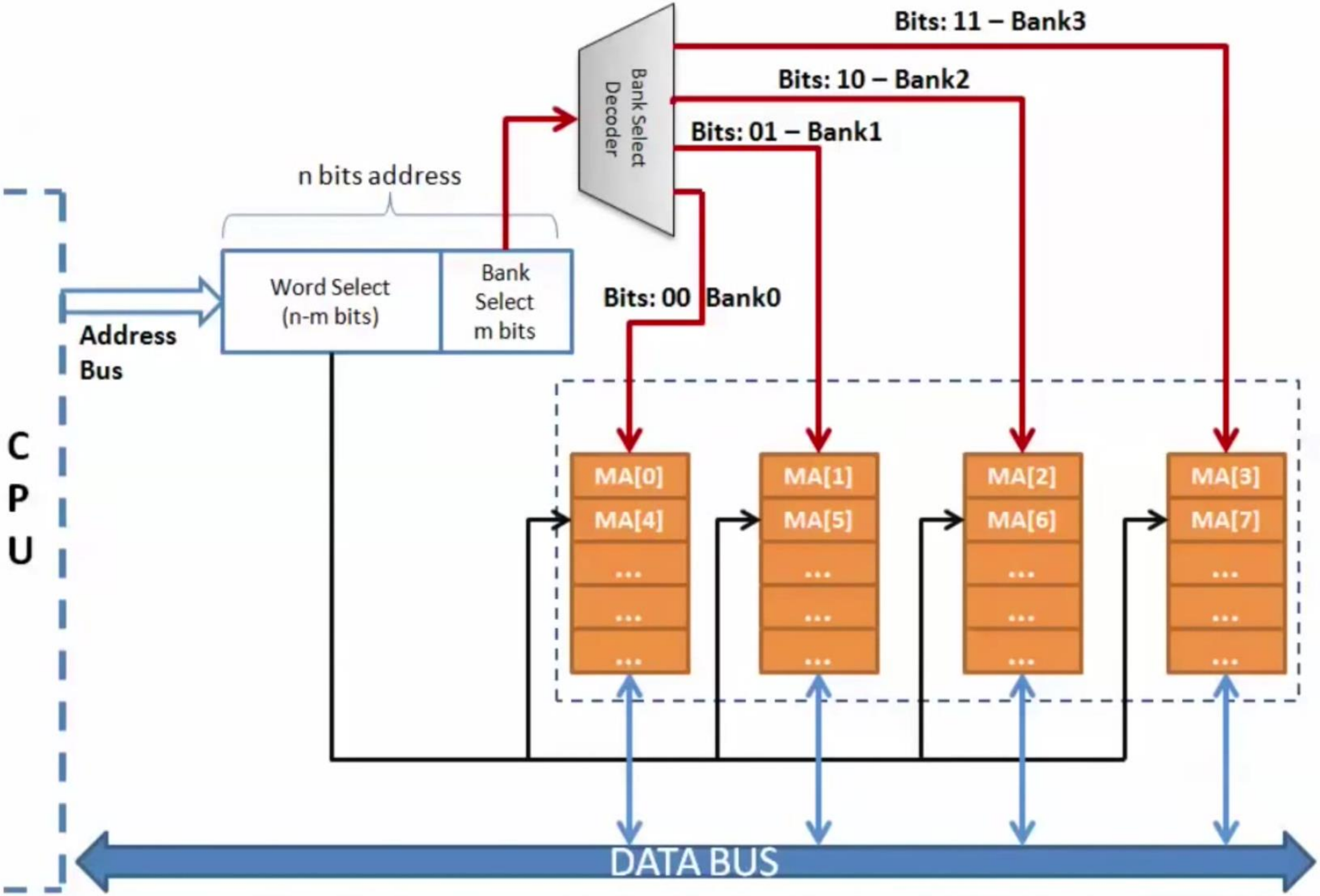
Memory Unit 0	Memory Unit 1	Memory Unit 2	Memory Unit 3
MA[0]	MA[1]	MA[2]	MA[3]
MA[4]	MA[5]	MA[6]	MA[7]
1	...	...	...

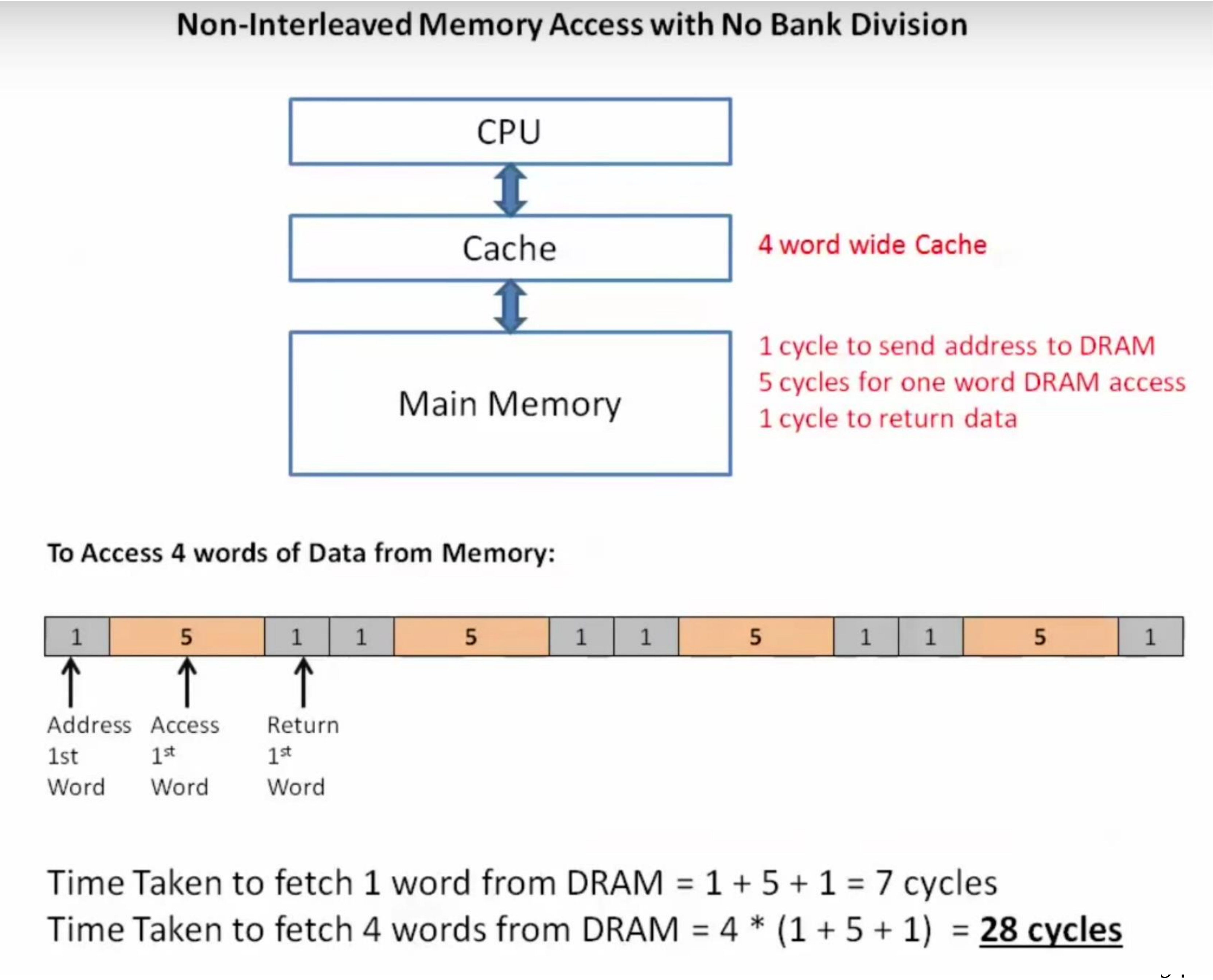
Access Pattern without Memory Interleaving

Access Pattern with Memory Interleaving

Data 1
Data 2
Reduced Access Time
Start Access Memory Unit 1 for Data1
Start Access Memory Unit 2 for Data 2

Abstract Architecture





Non-Interleaved Memory Access with No Bank Division

4 word wide Cache

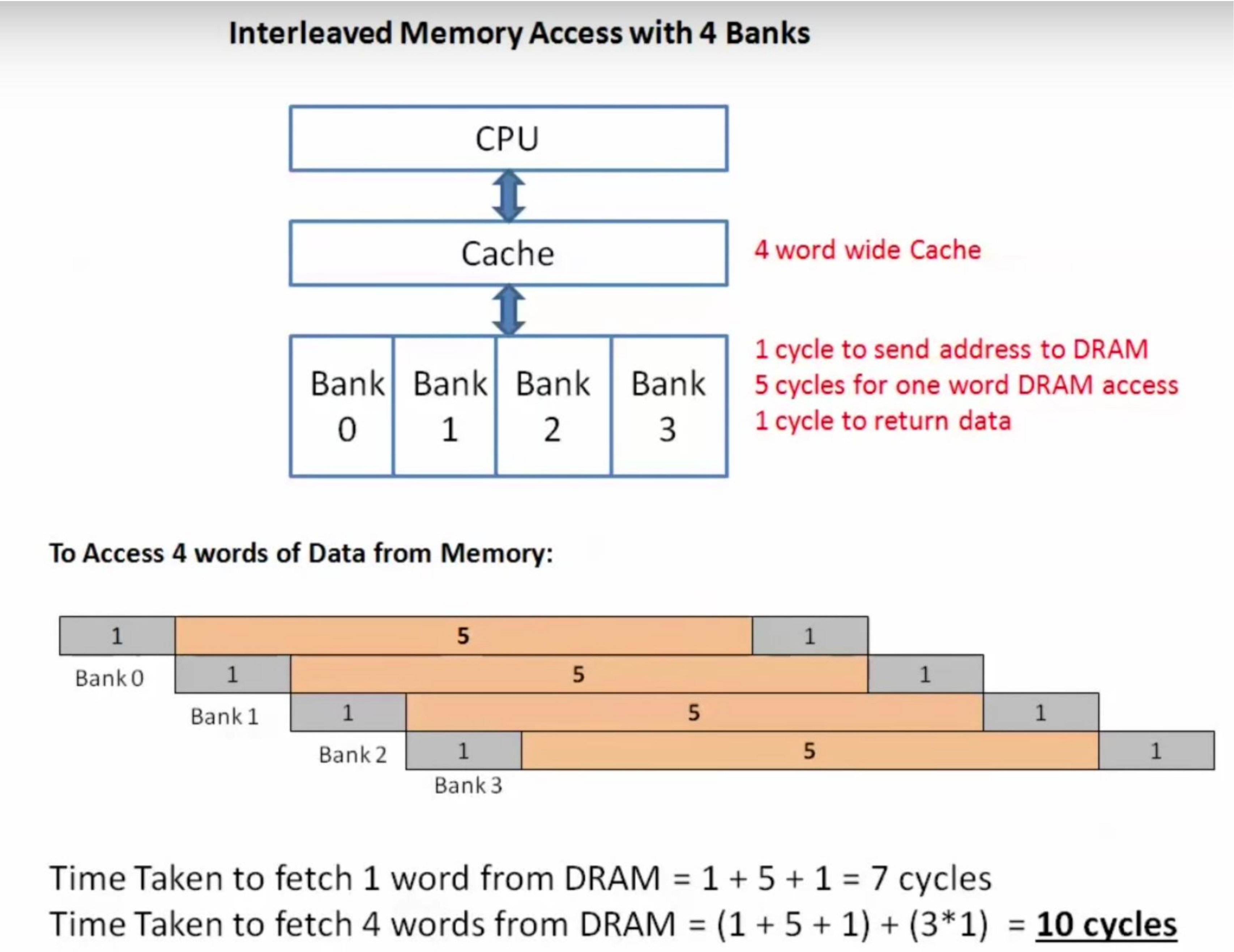
1 cycle to send address to DRAM
5 cycles for one word DRAM access
1 cycle to return data

To Access 4 words of Data from Memory:

1	5	1	1	5	1	1	5	1	1	5	1

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**



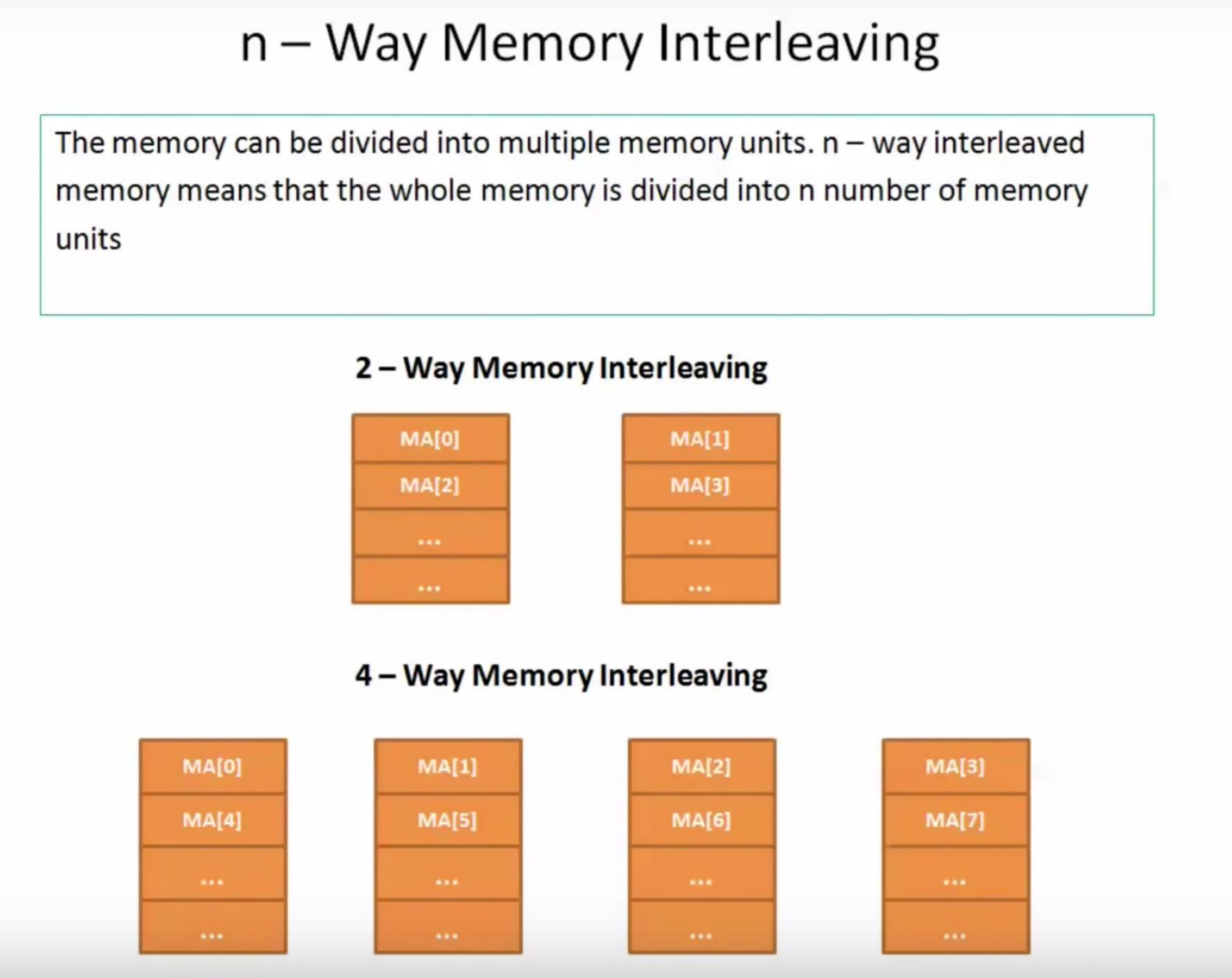
Interleaved Memory Access with 4 Banks

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

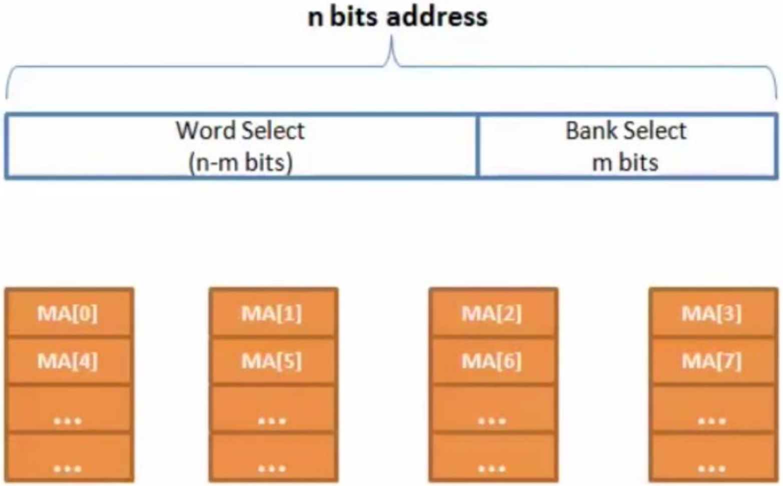
MA[0]	MA[1]
MA[2]	MA[3]
...	...
...	...

4 – Way Memory Interleaving

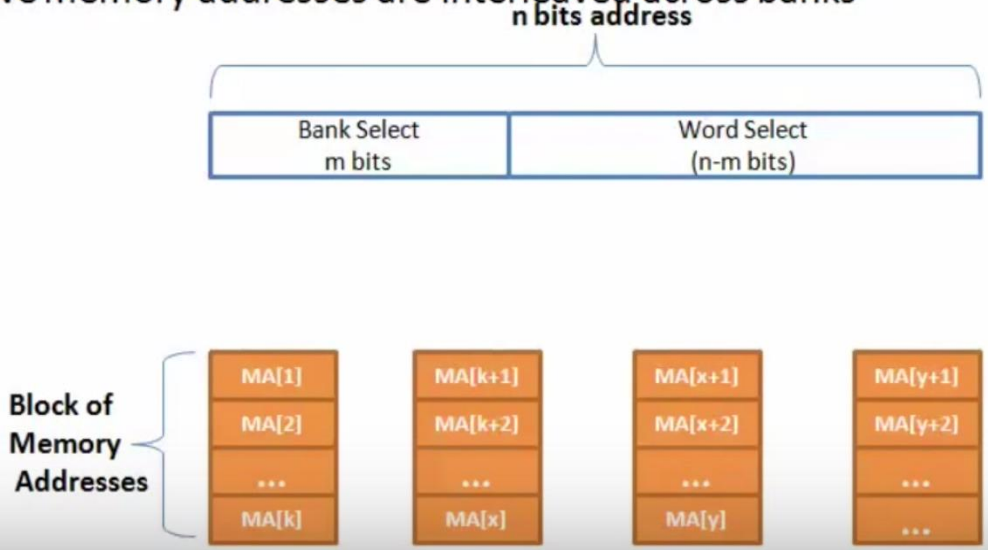
MA[0]	MA[1]	MA[2]	MA[3]
MA[4]	MA[5]	MA[6]	MA[7]
...	...	...	...
...	...	...	...

Types of Memory Interleaving

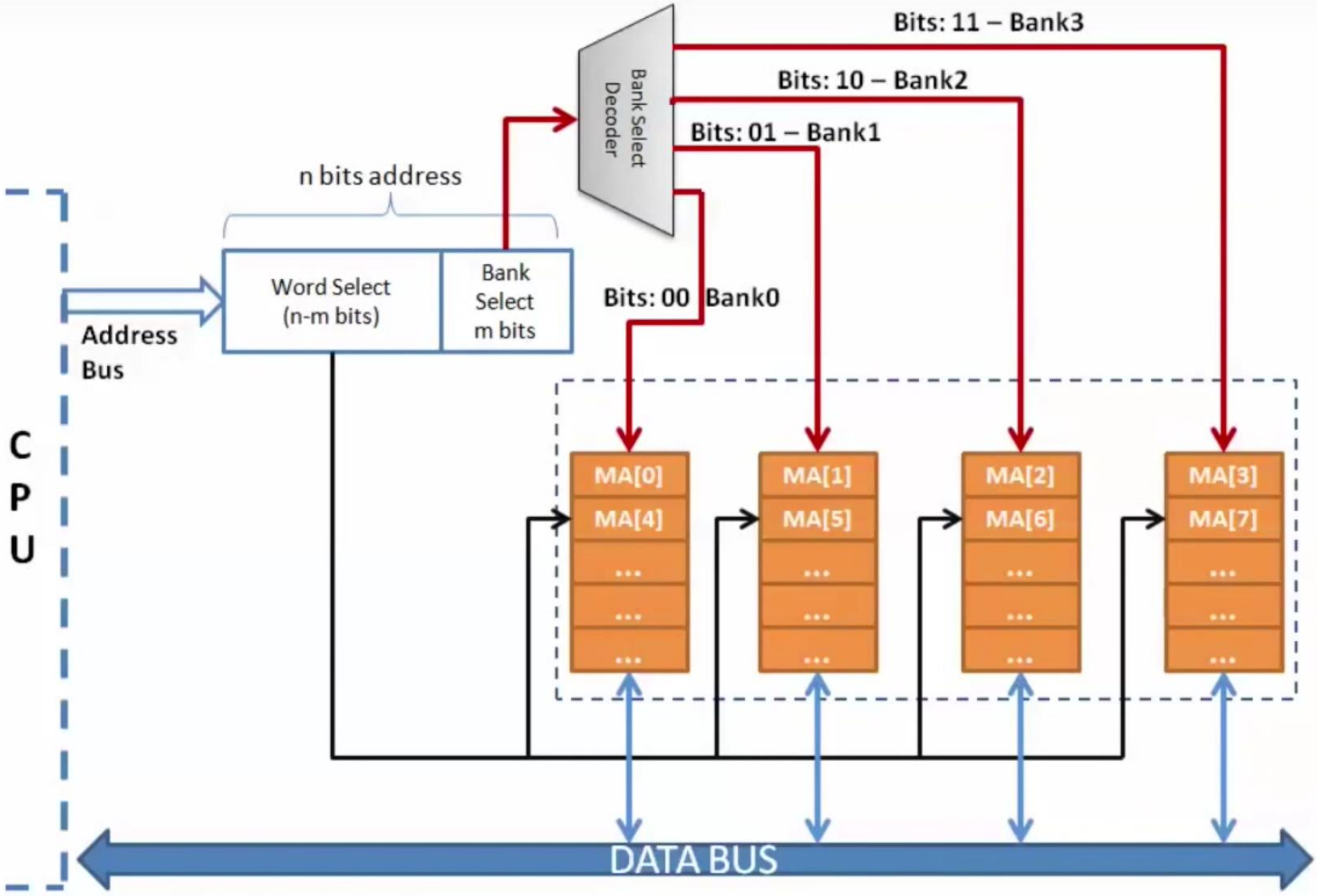
1. **Low Order Interleaving** : Least significant address bits are used for bank select. All successive memory addresses are interleaved across banks



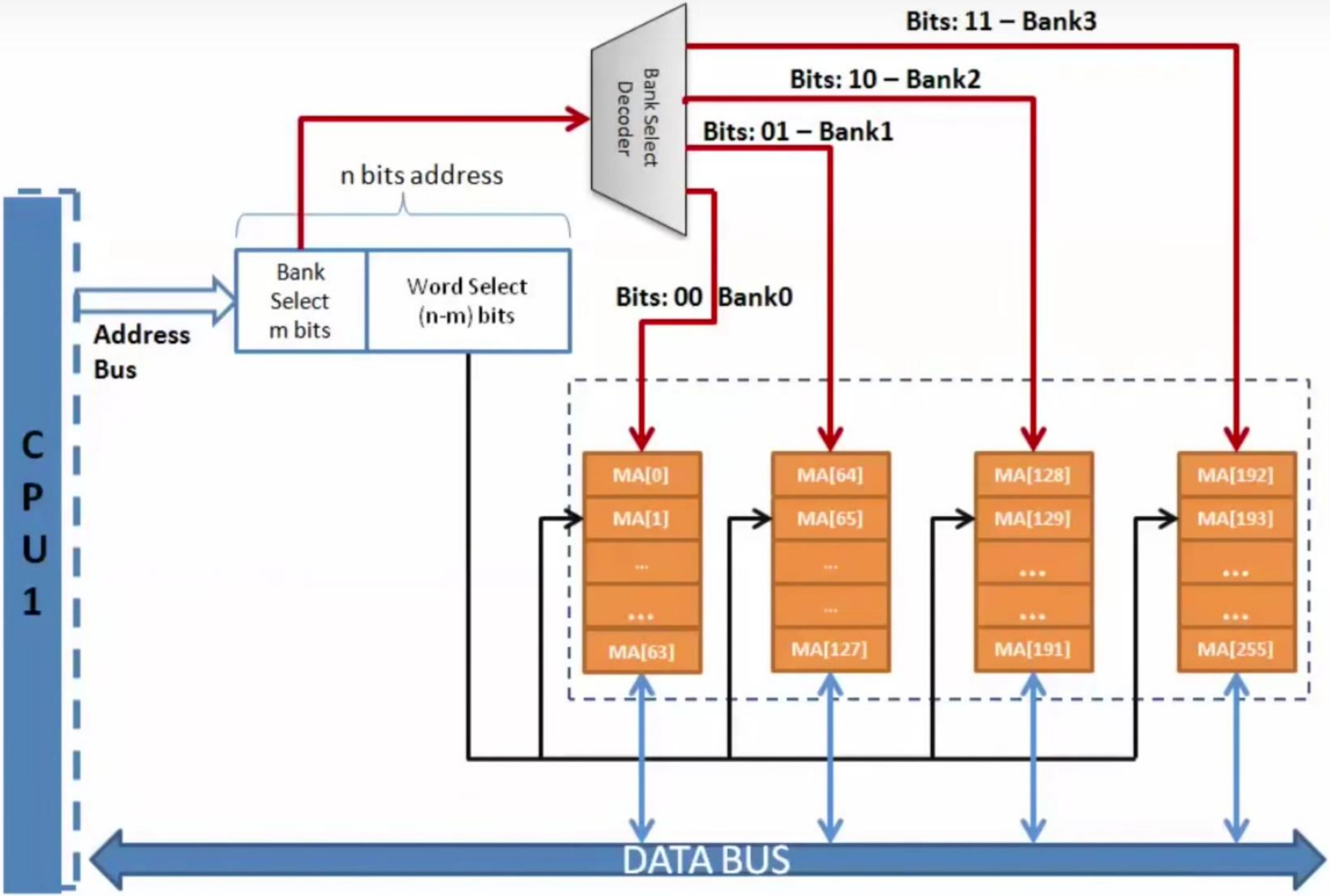
2. **High Order Interleaving** : Most significant address bits are used for bank select. Block of successive memory addresses are interleaved across banks



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;