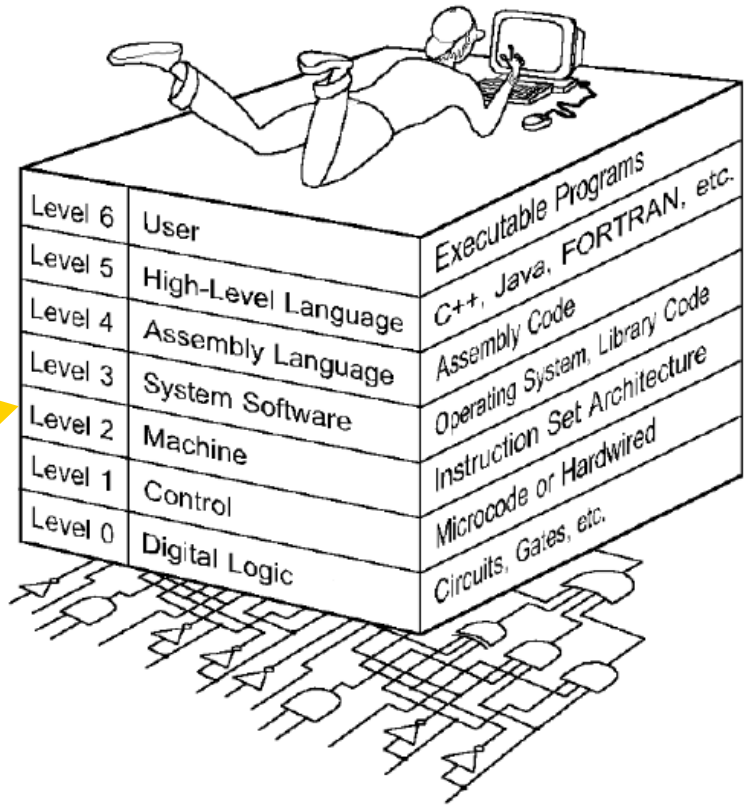


Chapter 6 – Memory

LO-5:

- Design simple **memory devices** and **system-level memories**.

>>> Quiz-4 and **Test-4** (with Module-8 on x86 Architecture)

Chapter 6 Objectives & Topics

- Master the concepts of hierarchical memory organization.
- Understand how each level of memory contributes to system **performance**, and how the performance is measured.
- Master the concepts behind cache memory, virtual memory, memory segmentation, paging and address translation.

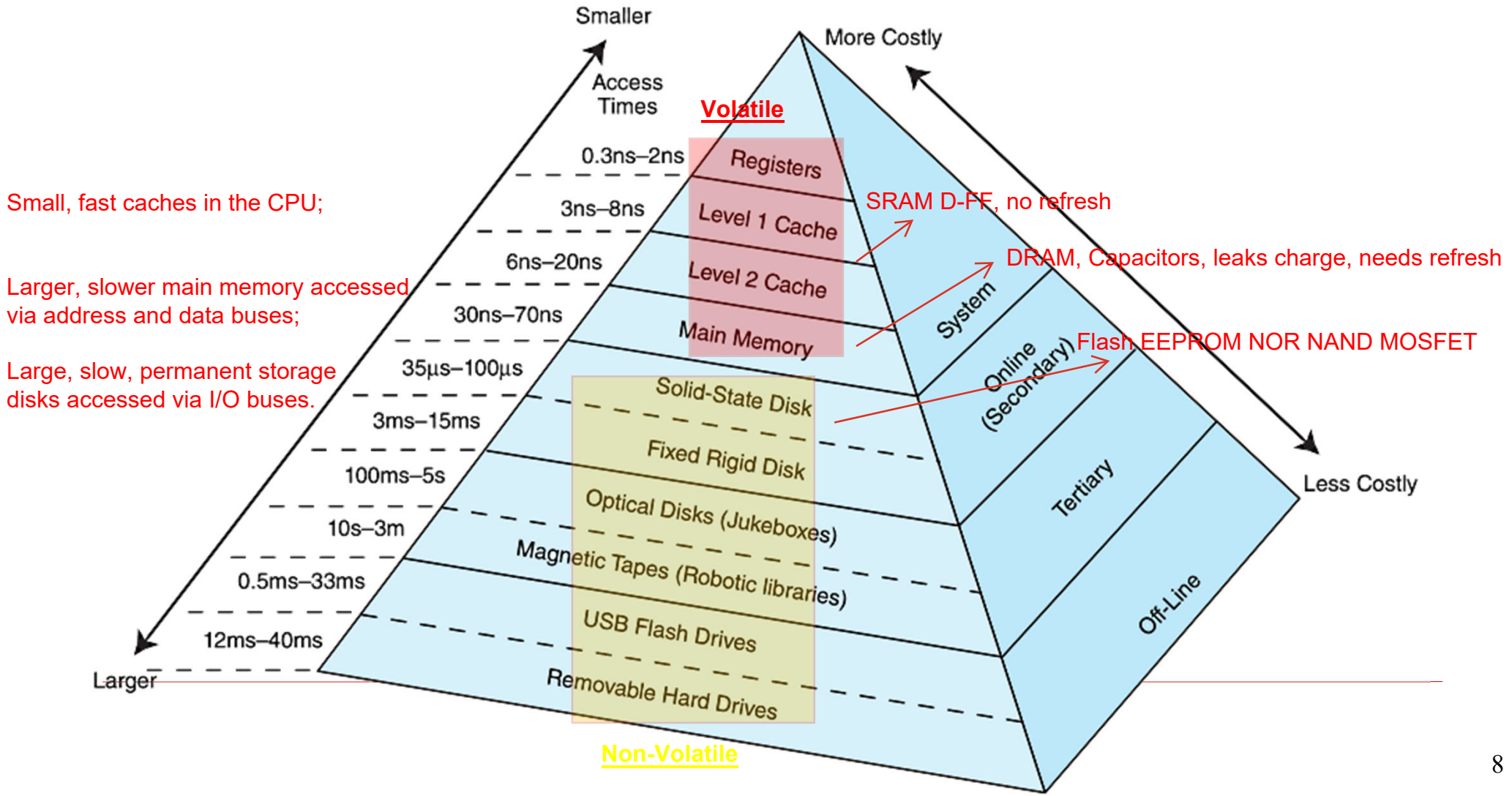
- **Types** of memory
- memory **hierarchy**
- **Cache** memory
- **Virtual** memory

- Memory is central to stored-program computer.
- Main Memory is built from DFFs, and is accessed by various ISAs using different addressing modes.

- Lets focus on memory organization, essential for the system performance.

6.3 Memory Hierarchy

- Memory is organized hierarchically for **best performance at the lowest cost**:



6.3 Memory Hierarchy

Register and Cache data transfers are hardwired and simple, that provides fast speed.

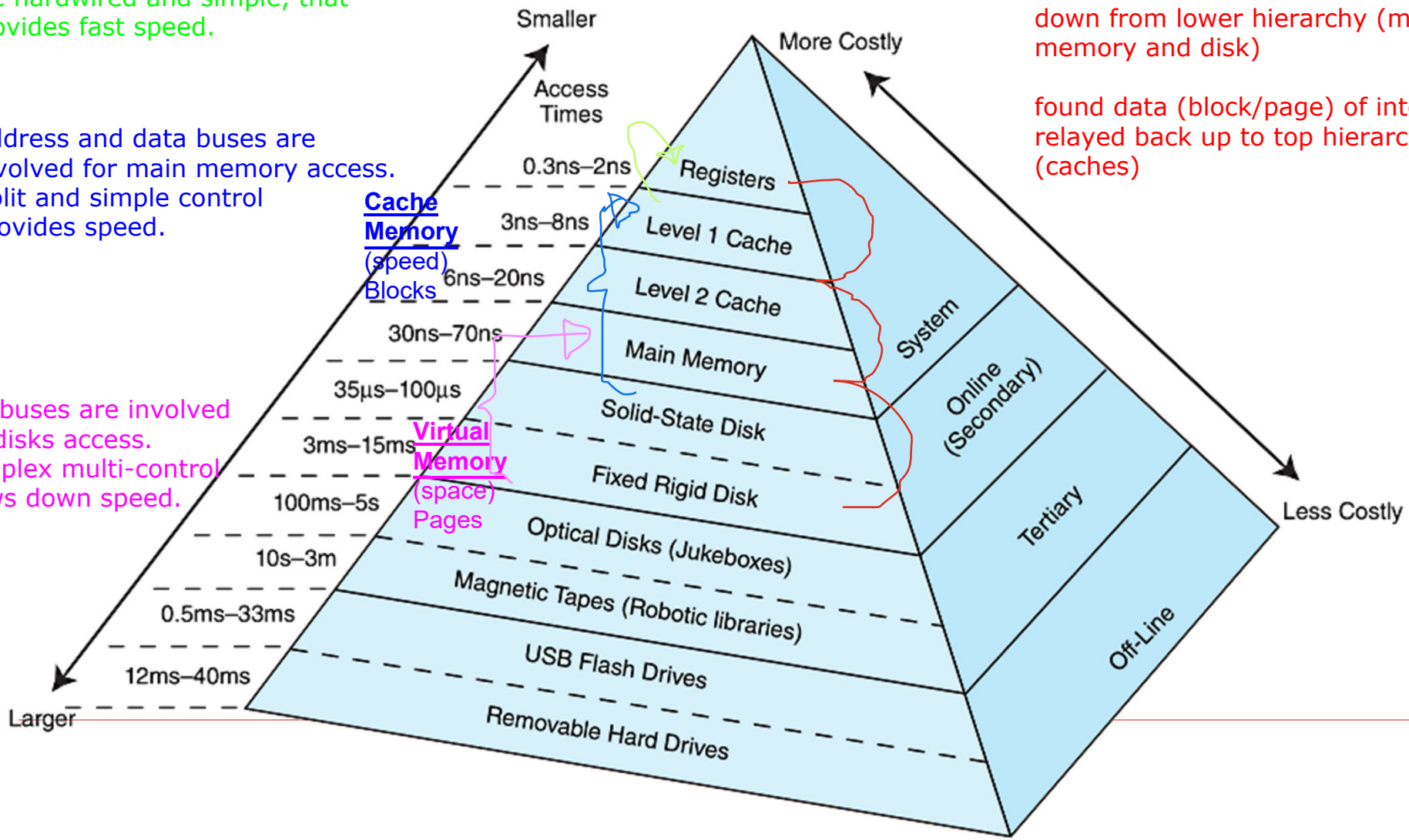
Address and data buses are involved for main memory access. Split and simple control provides speed.

I/O buses are involved for disks access. complex multi-control slows down speed.

Data loaded from cache to registers

unavailable data of interest is queried down from lower hierarchy (main memory and disk)

found data (block/page) of interest is relayed back up to top hierarchy (caches)

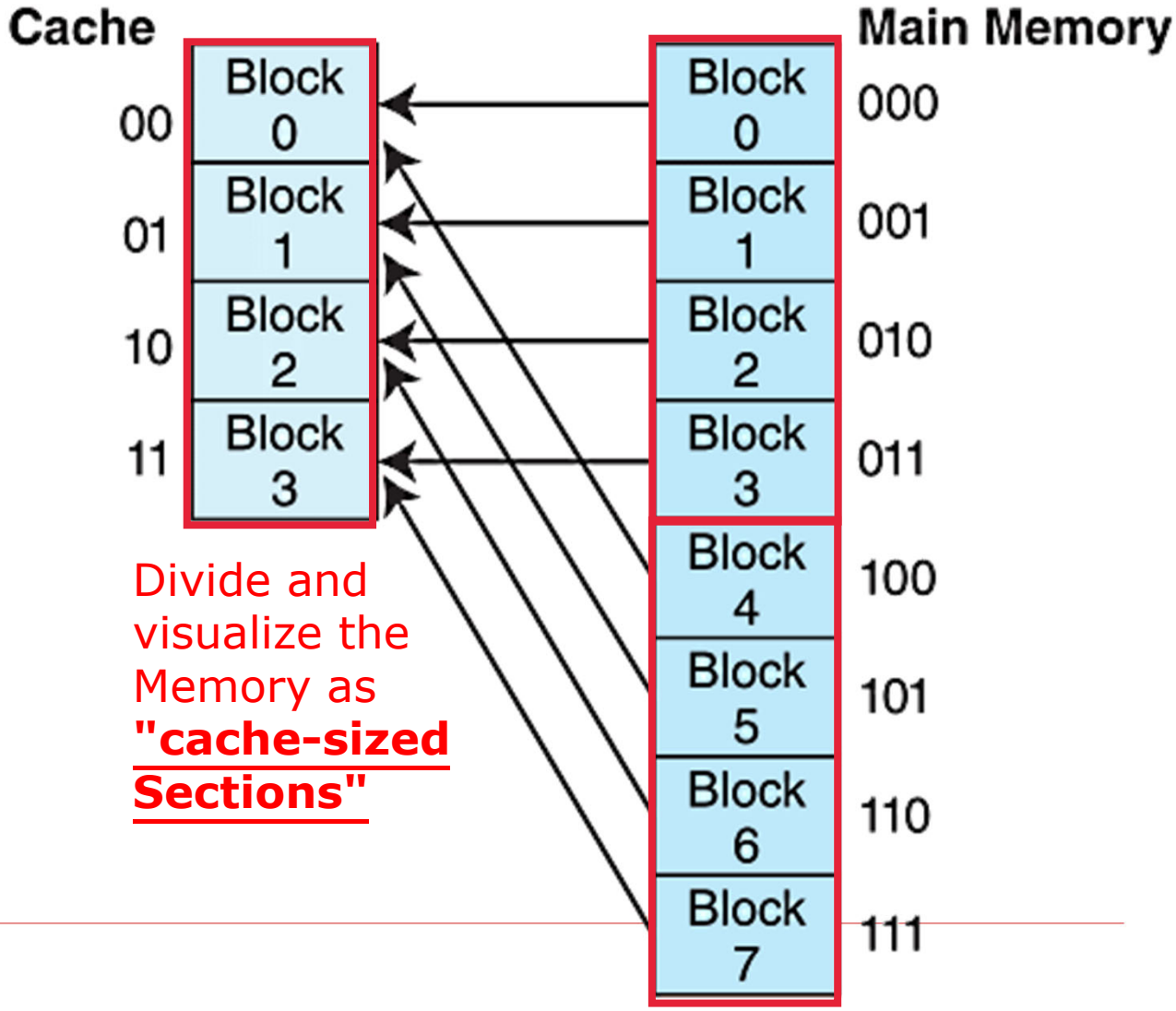


6.4 Cache Memory

- The purpose of cache memory is to **speed up** accesses by storing **recently used data closer** to the CPU, instead of storing it in main memory.
- Although cache is much smaller than main memory, its access time is a **fraction** of that of main memory.
- Three types of cache:
  - Direct mapped cache
  - Fully associative cache
  - Set associative cache

6.4 Cache Memory: Direct Mapped

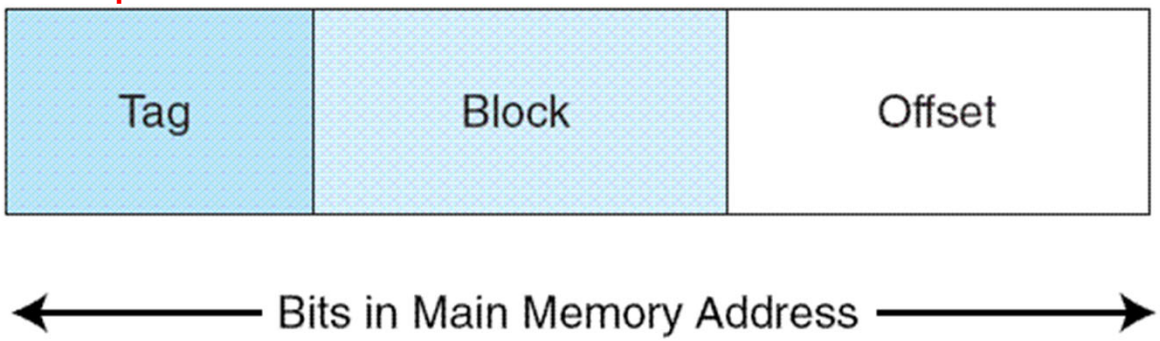
- cache block ***Y*** **= *X* mod *N***.
- one to one cache mapping



6.4 Cache Memory: Direct Mapped

- To perform direct mapping, the binary main memory address is partitioned into the *fields* **shown below**.
  - The *offset* field uniquely identifies an address **within a specific block**.
  - The *block* field **selects a unique block** of cache.
  - The *tag* field is whatever is **left over address**.

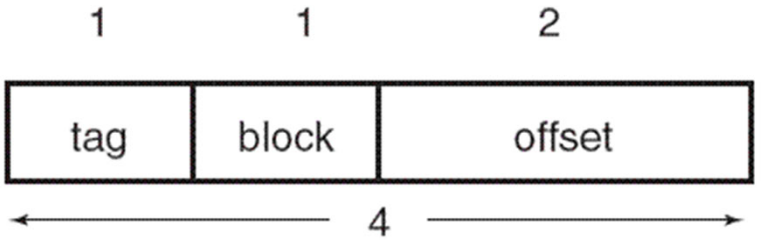
tag = "equivalent # of caches"

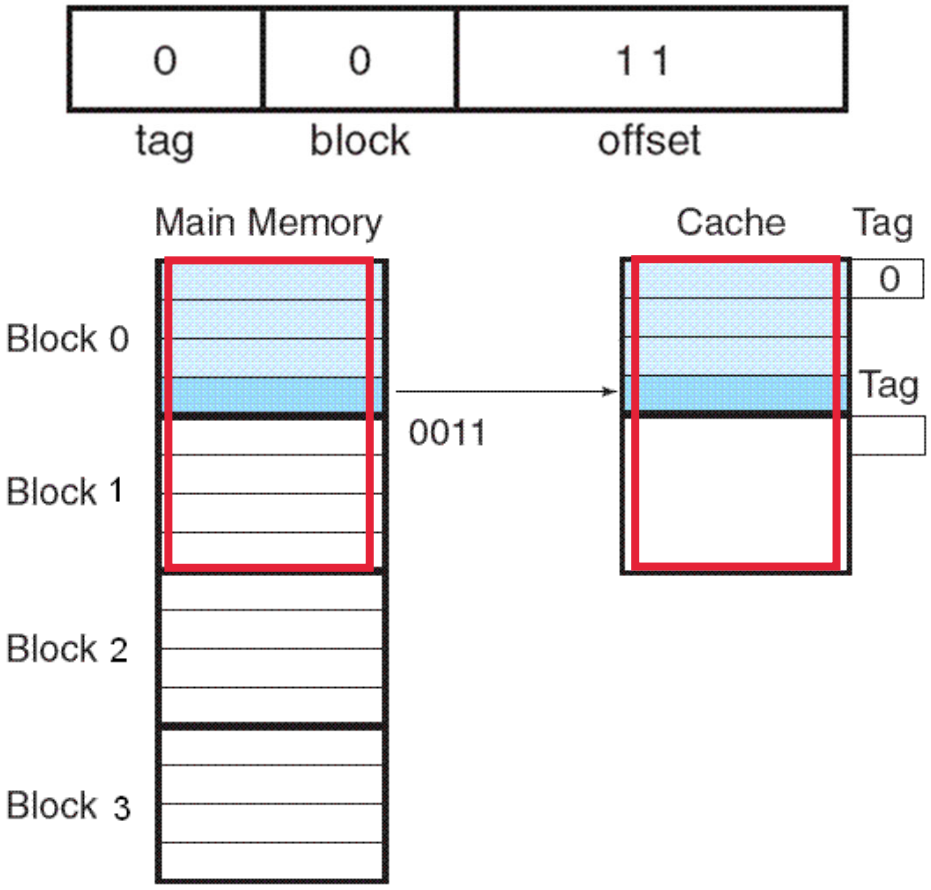


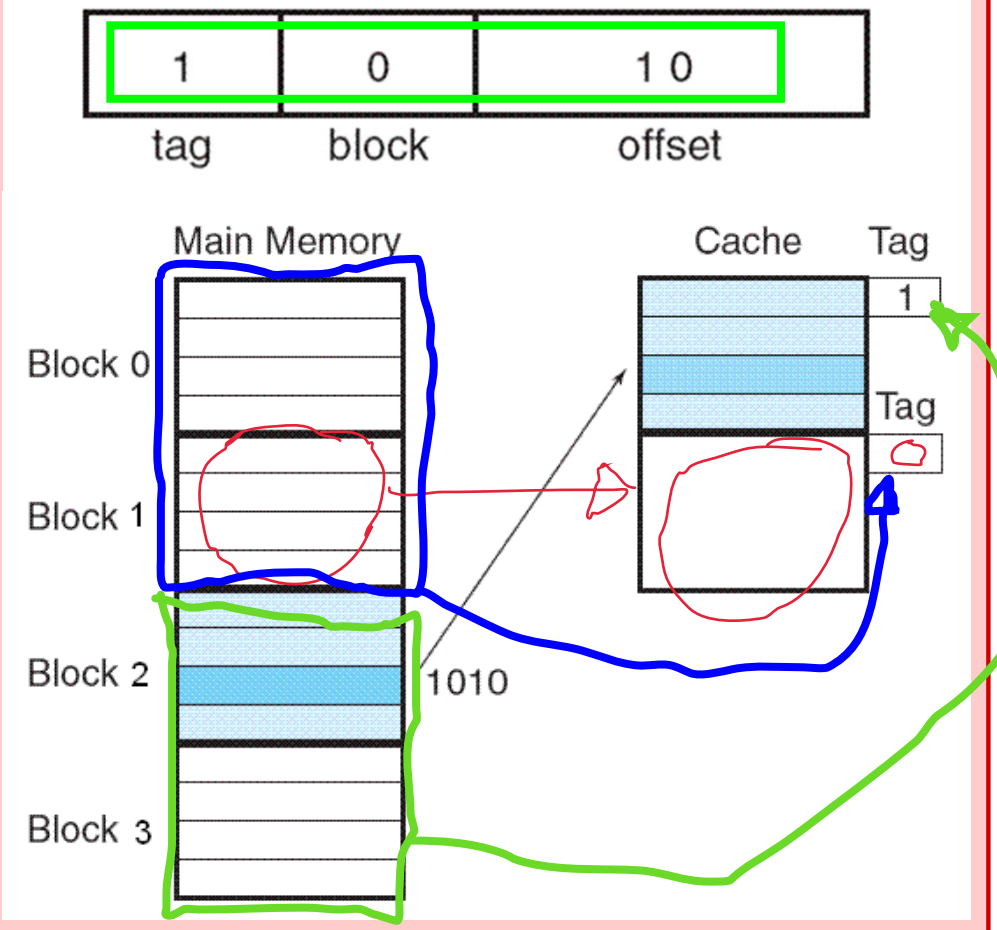
- The sizes of these fields are determined by characteristics of both memory and cache.

Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.

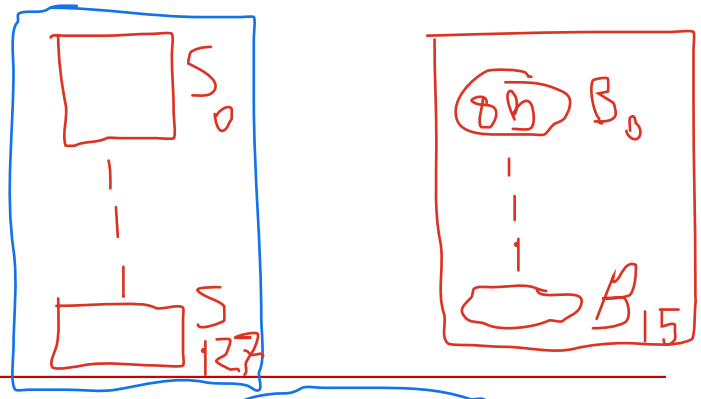
6.4 Cache Memory: Direct Mapped



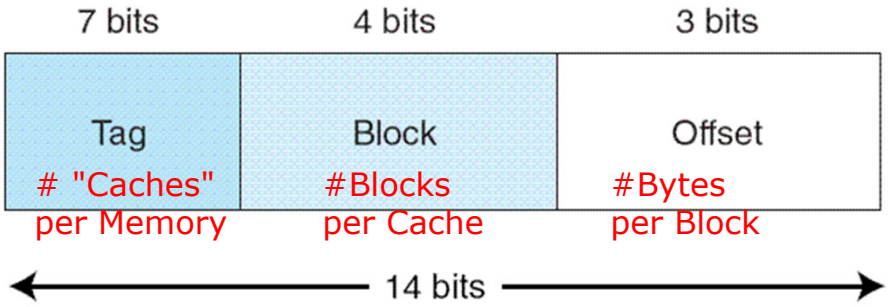


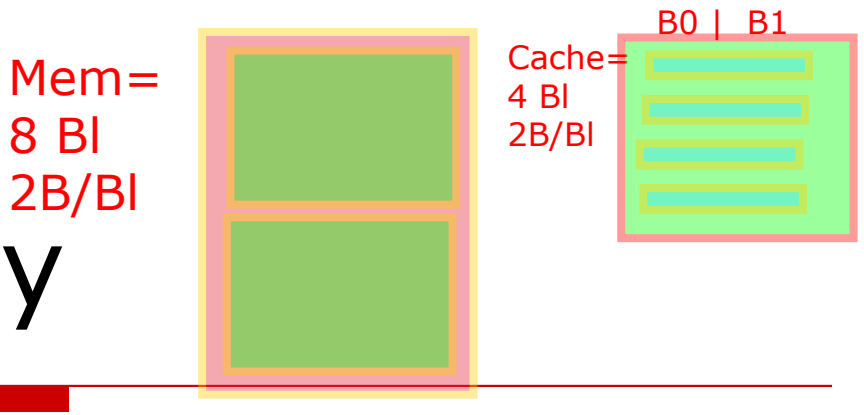


6.4 Cache Memory



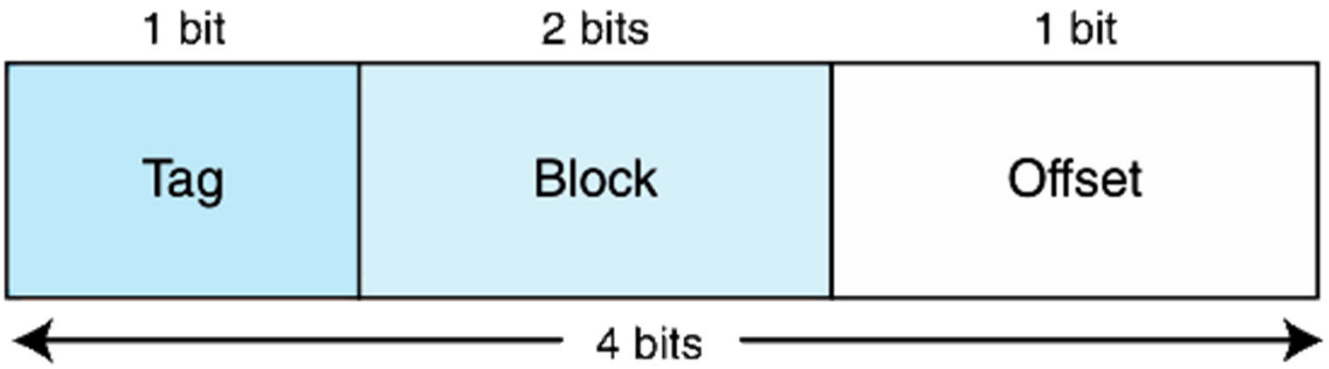
- EXAMPLE 6.2 Assume a byte-addressable memory consists of 2^{14} bytes, cache has 16 blocks, and each block has 8 bytes.
  - The number of memory blocks are: $\frac{2^{14}}{2^3} = 2^{11}$
  - Each main memory address requires14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the offset field
  - We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits.
  - The remaining 7 bits make up the tag field.





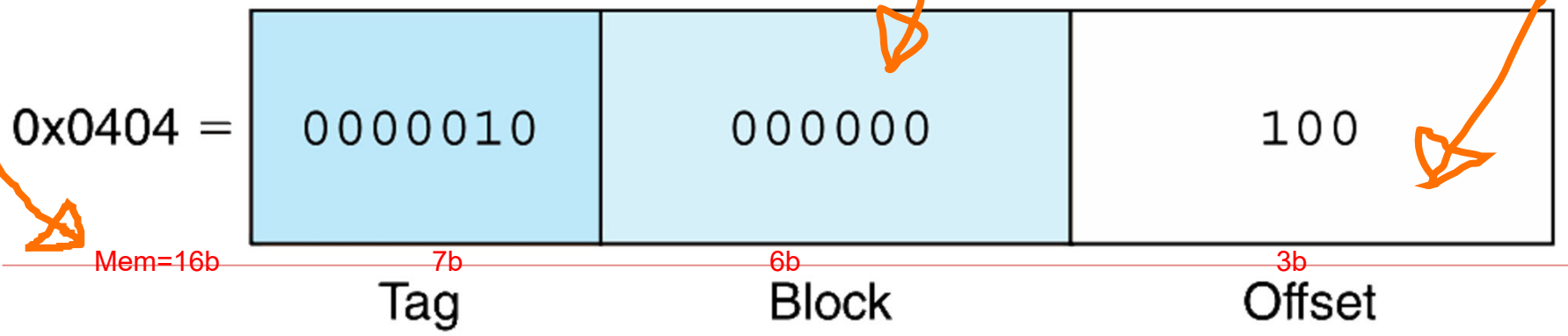
6.4 Cache Memory

- EXAMPLE 6.3 Assume a byte-addressable memory consisting of 16 bytes divided into 8 blocks. Cache contains 4 blocks. We know:
  - A memory address has 4 bits.
  - The 4-bit memory address is divided into the fields below.



6.4 Cache Memory

- EXAMPLE 6.4 Consider 16-bit memory addresses and 64 blocks of cache where each block contains 8 bytes. We have:
  - 3 bits for the offset
  - 6 bits for the block
  - 7 bits for the tag.
- A memory reference for 0x0404 maps as follows:

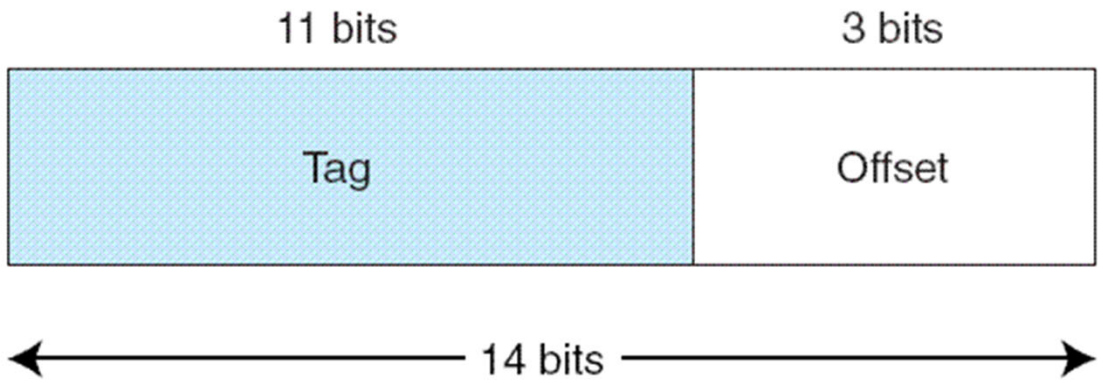


6.4 Cache Memory: Fully Associative

- Suppose **instead of** placing memory blocks in **specific cache locations** based on memory address, we could **allow a memory block to go** anywhere in cache.
- In this way, **cache would have to fill up before any block in cache is** evicted.
- A block's eviction is straightforward in direct mapped cache, whereas FA Cache uses an algorithm to determine the **victim** block.
- A memory address is partitioned into **only two fields: the tag and the word.**

6.4 Cache Memory: Fully Associative

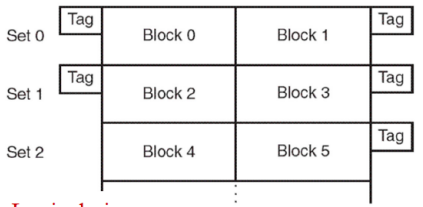
- Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:



- When the **cache is searched**, **all tags are searched in parallel** to retrieve the data quickly.
- This **requires special, costly hardware**.

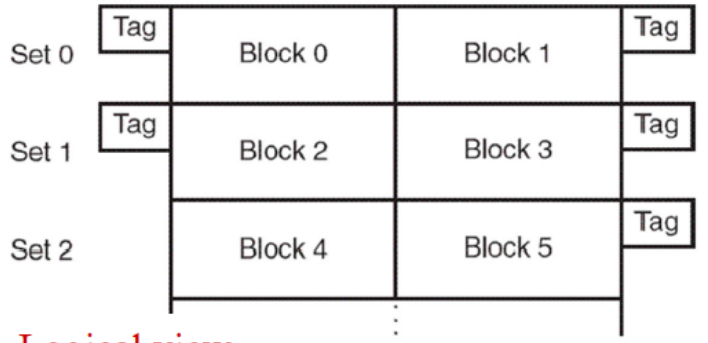
6.4 Cache Memory: Set Associative

- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An *N*-way set associative cache mapping is like direct mapped cache in that **a memory reference maps to a particular location** in cache.
- Unlike direct mapped cache, a memory reference **maps to a set of several cache blocks**, similar to the way in which fully associative cache works.
- Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.

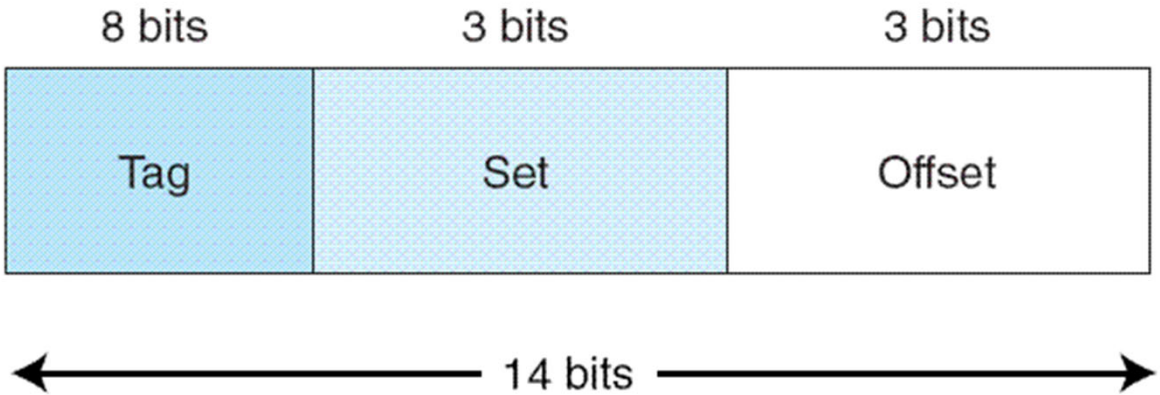


Logical view

6.4 Cache Memory



- EXAMPLE 6.5 Suppose we are using 2-way set associative mapping with a word-addressable main memory of 2^{14} words and a cache with 16 blocks, where each block contains 8 words.
  - Cache has a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache.
  - Thus, the set field is 3 bits, the offset field is 3 bits, and the tag field is 8 bits.



6.4 Cache Memory: Set Associative

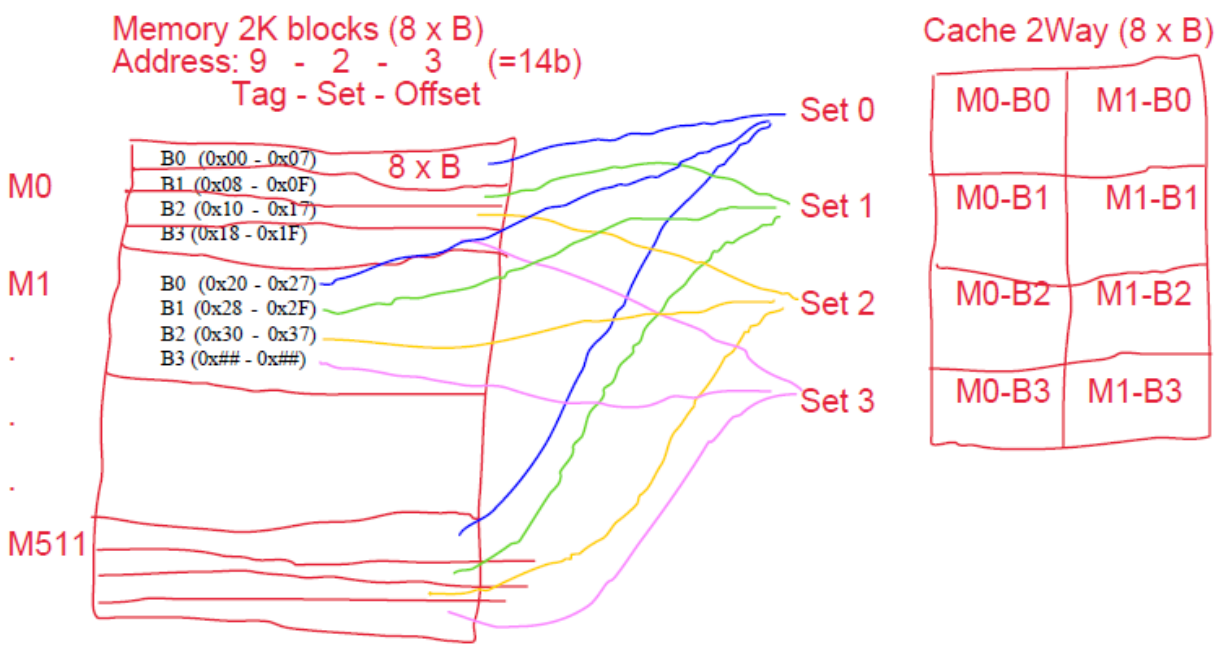
- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and offset.
- As with direct-mapped cache, the offset field chooses the word within the cache block, and the tag field uniquely identifies the memory address.
- The set field determines the set to which the memory block maps.
- HW Problem-8:

(a) A 2-way set-associative cache consists of four sets. Main memory contains 2K blocks of 8 bytes each and byte addressing is used.

(b) hit ratio for a program that loops three times from addresses 0x08 to 0x33 in main memory

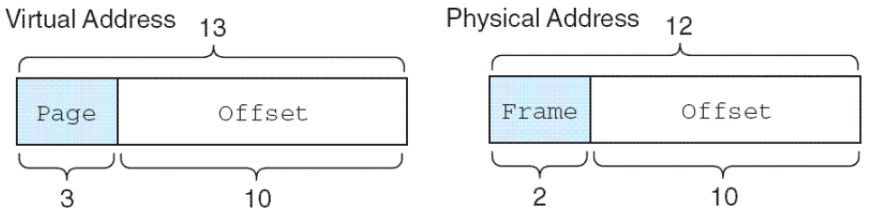
Loop-1: 6 misses, 5x7+3 = 38 hits
Loop-2: 0 misses, 5x8+4 =44 hits
Loop-3: 0 misses, 5x8+4 =44 hits

Hit Ratio = 126/132=95.5%



Summary: Cache

- Direct Mapped (1-1)
  - a specific memory block --> one cache block
  - eviction is easy
  - content search is easy
- Fully Associative (1-any)
  - a memory block --> any cache block (full scope)
  - eviction algorithm needed
  - content search is exhaustive, parallel, costly
- Set Associative (1-set)
  - a memory block --> one set (of blocks) in cache
  - eviction algorithm is simpler but needed
  - content search is limited

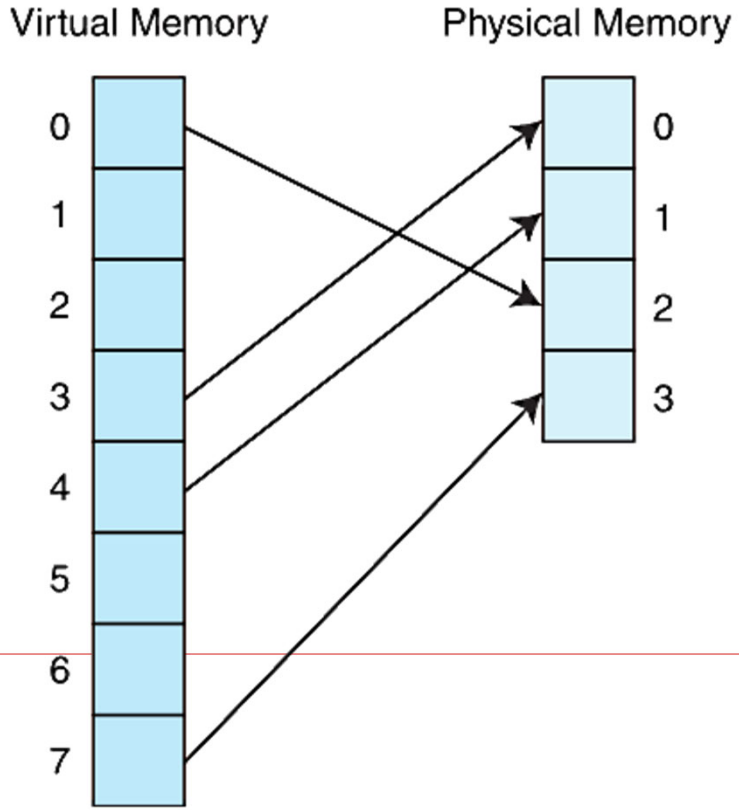


6.5 Virtual Memory

- Cache memory is for faster data access, virtual memory for increasing **capacity by extending** main memory using disk drive space.
- Virtual memory **maps** the main memory page frames to disk **pages (same size)** when not needed.
- A process parts' can be on both disk and main memory **non-contiguously** where unnecessary pages are in slower disk storage.
- Programs create **virtual addresses** that are mapped to physical addresses by the **memory manager of operating system**. A physical address is actual physical memory address.

6.5 Virtual Memory

- Information concerning the location of each page, whether on disk or in memory, is maintained in a data structure called a *page table* (shown below).
- There is one page table for each active process.



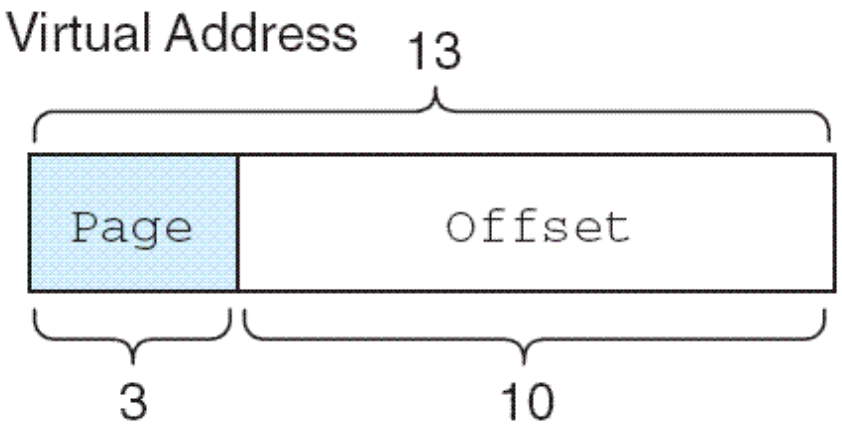
Logical Page

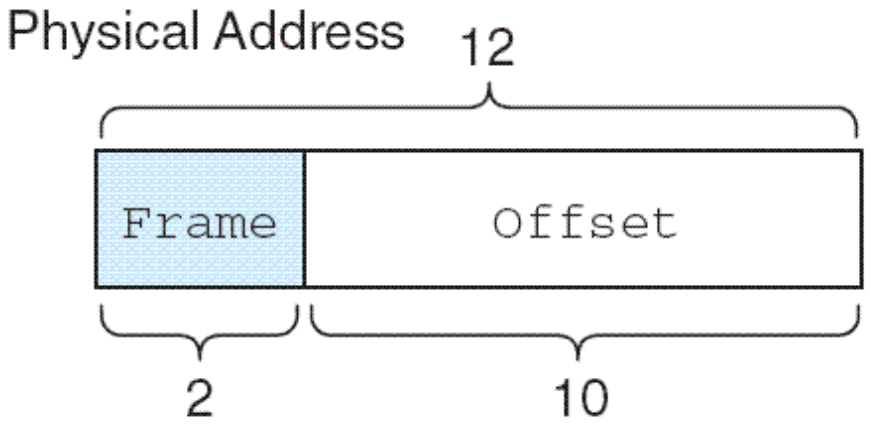
Page Table

	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

6.5 Virtual Memory

- As an example, suppose a system has a virtual address space of 8K and a physical address space of 4K, page size is 1024 and the system uses byte addressing.
  - We have $2^{13}/2^{10} = 2^3$ virtual pages.
- A virtual address has 13 bits (8K = 2^{13}) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first two bits for the page frame and the trailing 10 bits the offset.





Continued ....

6.5 Virtual Memory

continued from previous example

- Suppose we have the page table shown below.
- What happens when CPU generates virtual address $5459_{10} = 1010101010011_2 = 0x1553$?

Page Table

Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Page	Addresses Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

@5459 is in the page that is in memory ==> No Page fault

Page faults occur when a logical address requires that a **page be brought in from disk**. An existing page in memory will be **evicted** and this space will be used for incoming one.

6.5 Virtual Memory

- What happens when the CPU generates address 100000000100_2?

Valid bit=0 ==> a Mem page evicted and replaced with one from disk!

Page Table

Page	Frame	Valid Bit
0	–	0
1	~~3~~	0 ~~1~~
2	0	1
3	–	0
4	~~–~~ 3	1 ~~0~~
5	1	1
6	2	1
7	–	0

say, victim is picked

Page Fault!

Addresses

Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

6.5 Virtual Memory

- The effective access time (EAT) takes all levels of memory into consideration.
- Thus, **virtual** memory is also a **factor** in the calculation, and we also have to **consider page table access time**.
- Suppose a main memory access takes 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk. We have:

  EAT = 0.99(200ns + 200ns) + 0.01(10ms) = 100.396us.

6.5 Virtual Memory

- Even if we had **no page faults**, the EAT would be 400ns because memory is always read twice: First to access the page table, and second to load the page from memory.
- Because **page tables** are read constantly, it makes sense to keep them **in a special cache** called a *translation look-aside buffer* (TLB).
- TLBs are a special associative cache that stores the mapping of virtual pages to physical page frames.

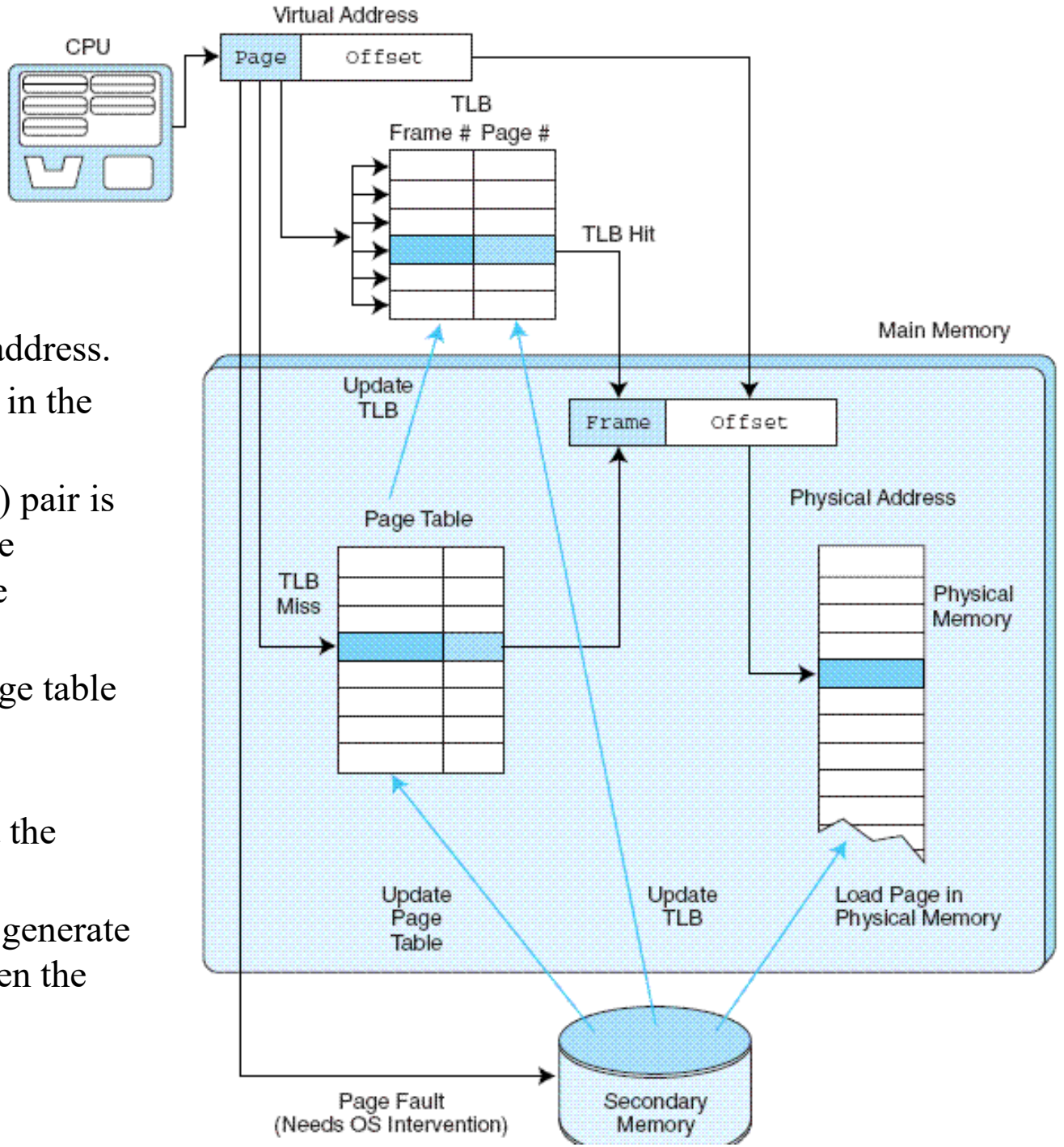
The next slide shows address lookup steps when a TLB is involved.

TLB lookup process

1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Search for the virtual page number in the TLB.
4. If the (virtual page #, page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
5. If there is a TLB miss, go to the page table to get the necessary frame number.

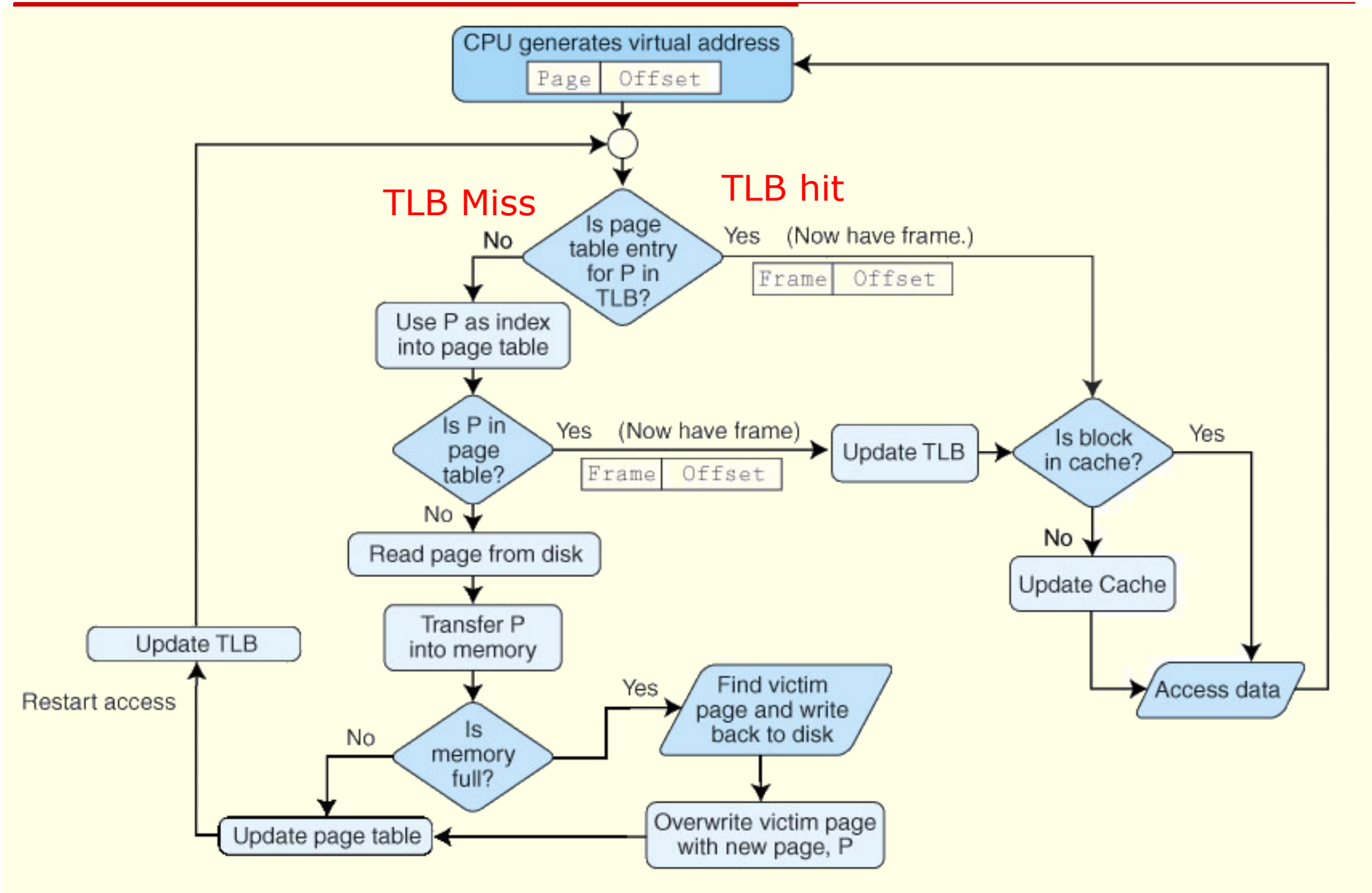
If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.

6. If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.



Putting it all together:
The TLB, Page Table,
and Main Memory

6.5 Virtual Memory



Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

6.5 Virtual Memory: Segmentation

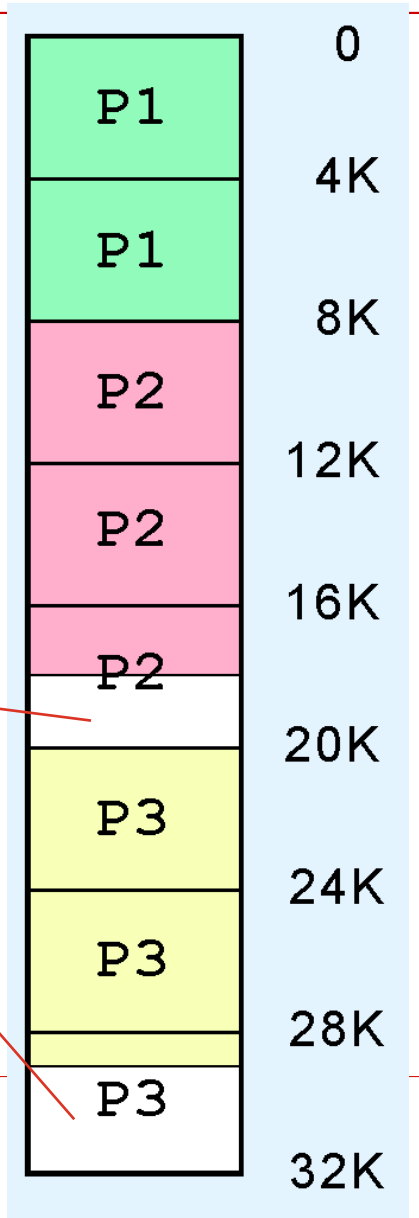
- Another approach to virtual memory is through *segments*, which can be allocated anywhere in memory.
- Instead of dividing memory into **equal-sized pages**, virtual address space is divided into variable-length **segments**, often under the **control of the programmer**.
- A segment is **located through its entry** in a segment table, **which contains** the segment's memory **location and a bounds** limit that indicates its size.
- After a page fault, the operating system searches for a location in memory large enough to hold the segment that is retrieved from disk.

Both **paging** and segmentation can cause fragmentation (small, unusable clusters).

6.5 Virtual Memory: Fragmentation

- Despite the fact that there are **enough free bytes** in memory to load the fourth process, **P4 has to wait** for one of the other three to terminate, because there are **no unallocated frames**.
- This is *internal* ***fragmentation inside the page frames***.

Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

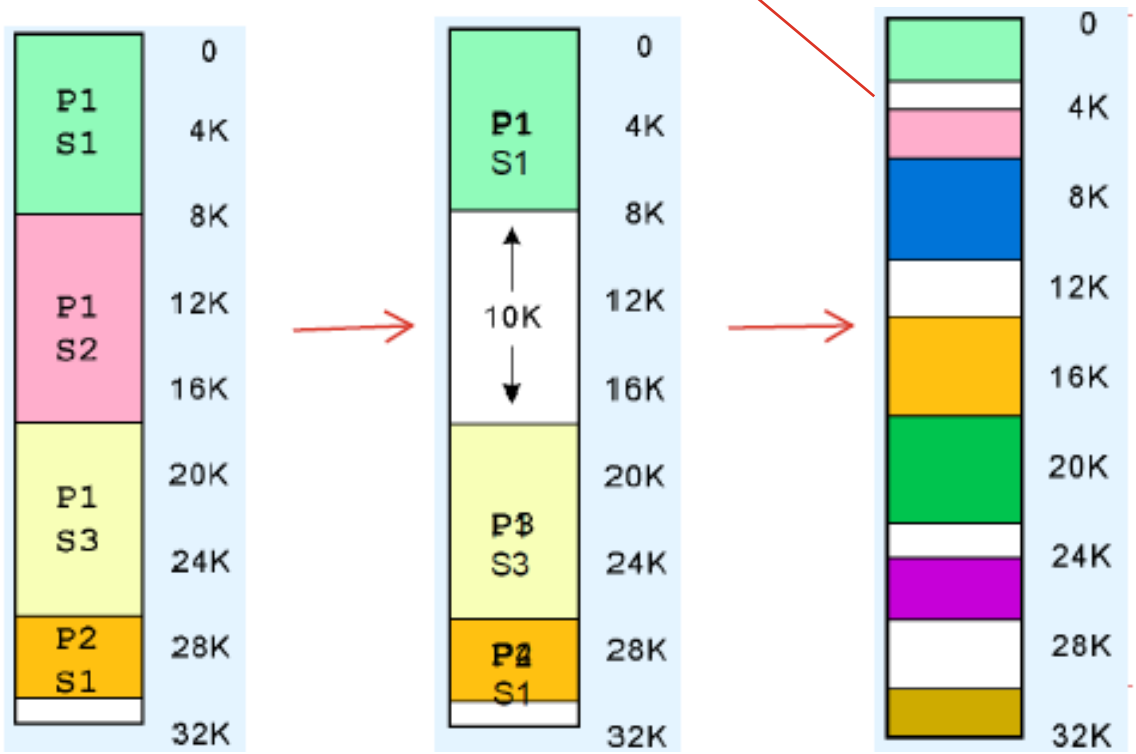


Both paging and **segmentation** can cause fragmentation (small, unusable clusters).

6.5 Virtual Memory: Fragmentation

- Only 1K free, so P2 S2 waits.
- P1S2 execution ended or evicted, but P1S2 still cant be loaded due to non-contiguous 11K space available.
- Over time, this causes **external fragmentation**.
- Eventually, this memory is recovered through **compaction**, and the process starts over.

Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



6.6 A Real-World Example

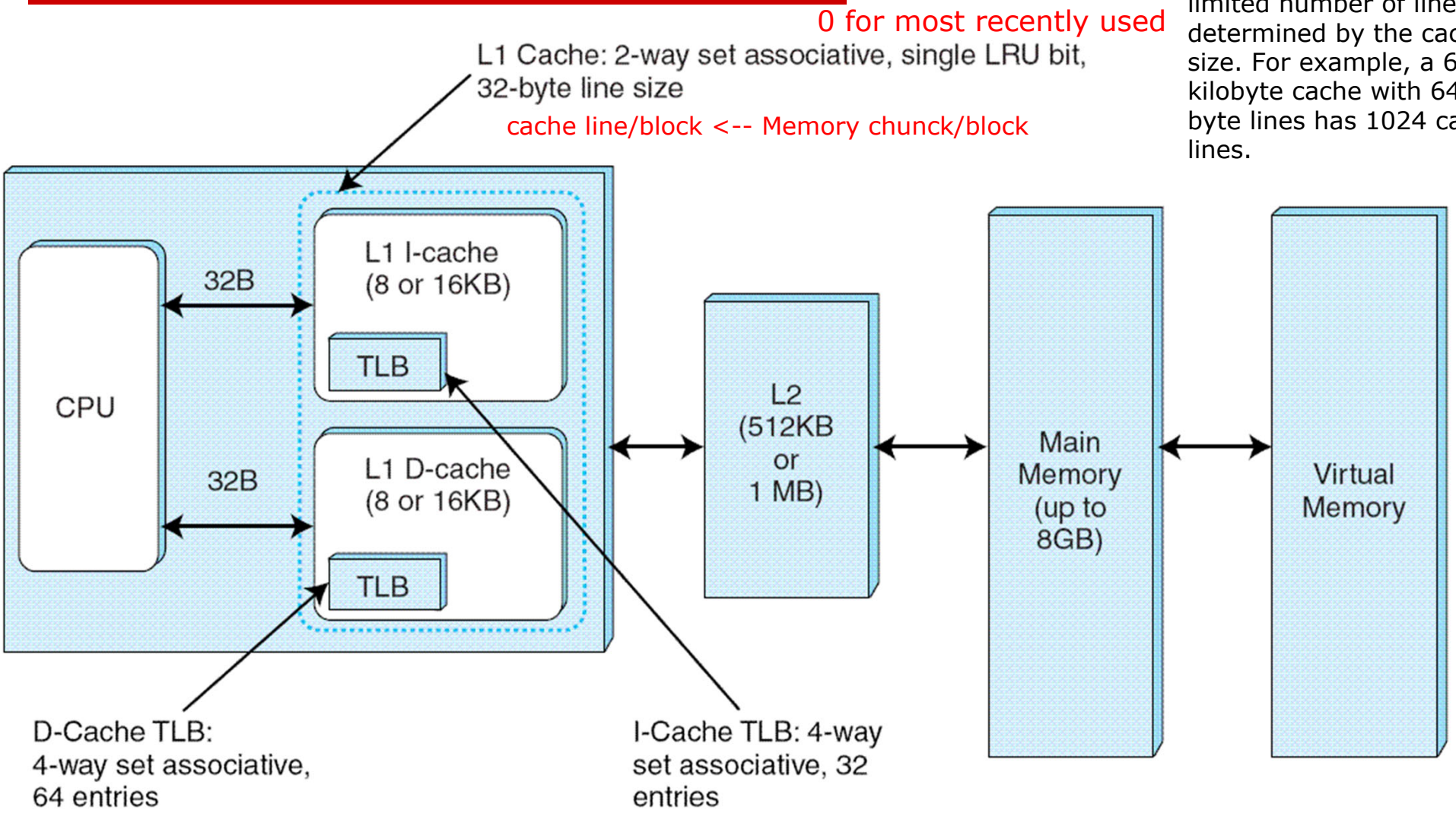
- The **Pentium** architecture supports both **paging and segmentation**, and they can be used **in various combinations** including unpaged unsegmented, segmented unpaged, and unsegmented paged.
- The processor supports two levels of cache (**L1 and L2**), both having a block size of **32 bytes**.
- The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- The L1 cache is in two parts: and instruction cache (**I-cache**) and a data cache (**D-cache**).

The next slide shows this organization schematically.

6.6 A Real-World Example

The chunks of memory handled by the cache are called cache lines. The size of these chunks is called the cache line size. Common cache line sizes are 32, 64 and 128 bytes.

A cache can only hold a limited number of lines, determined by the cache size. For example, a 64 kilobyte cache with 64-byte lines has 1024 cache lines.



Each TLB entry holds a virtual page number and its corresponding physical page frame number.