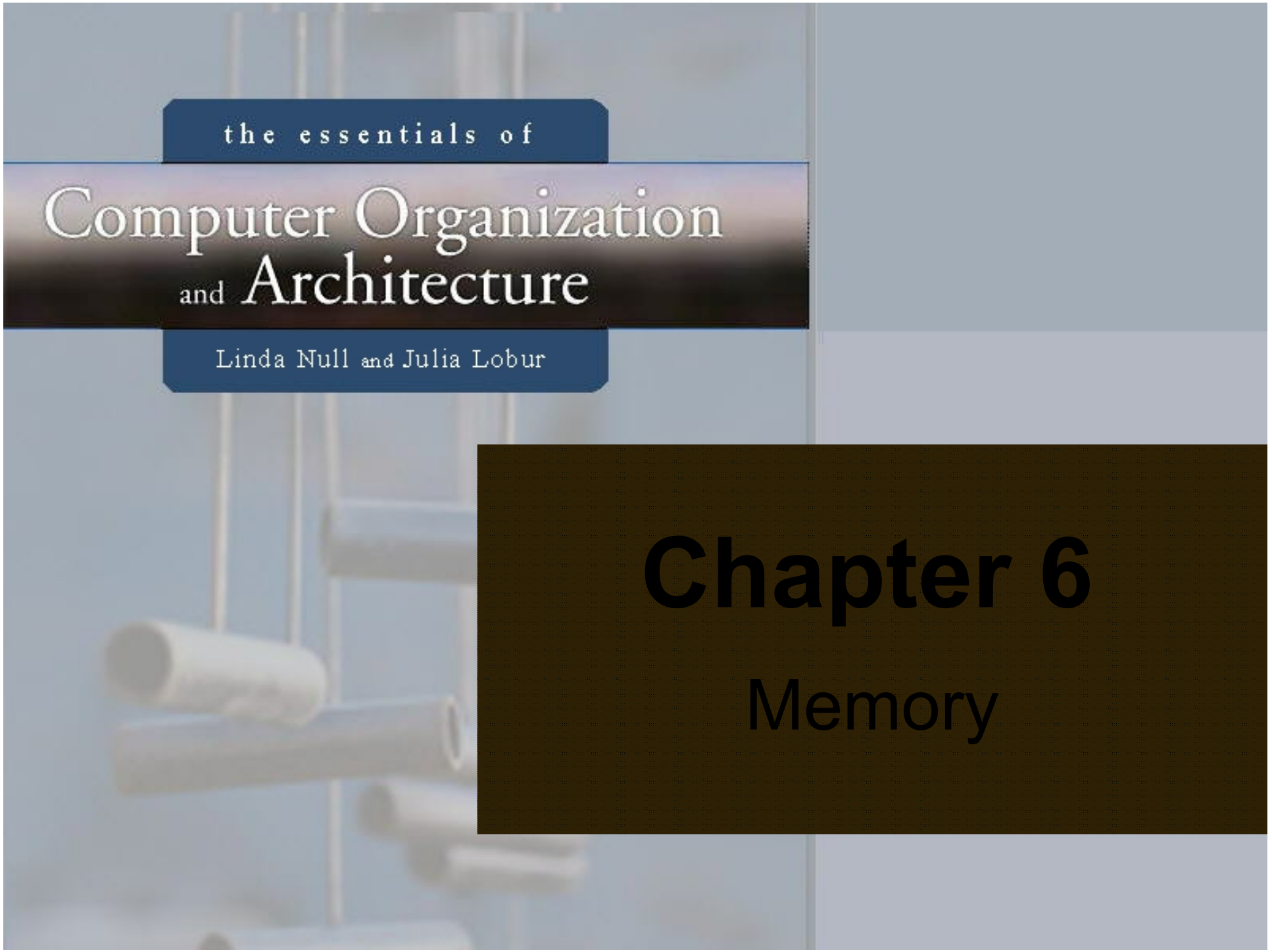


Chapter 6

Memory

Chapter 6 Objectives

- 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.

6.1 Introduction

- Memory lies at the heart of the stored-program computer.
- In previous chapters, we studied the components from which memory is built and the ways in which memory is accessed by various ISAs(Instruction System Architecture).
- In this chapter, we focus on memory organization. A clear understanding of these ideas is essential for the analysis of system performance.

6.2 Types of Memory

- There are two kinds of main memory: *random access memory, RAM, and read-only-memory, ROM*.
- There are two types of RAM, dynamic RAM (DRAM) and static RAM (SRAM).
- Dynamic RAM consists of capacitors that slowly leak their charge over time. Thus they must be refreshed every few milliseconds to prevent data loss.
- DRAM is “cheap” memory owing to its simple design.

6.2 Types of Memory

- SRAM consists of circuits similar to the D flip-flop that we studied in Chapter 3.
- SRAM is very fast memory and it doesn't need to be refreshed like DRAM does. It is used to build cache memory, which we will discuss in detail later.
- ROM also does not need to be refreshed, either. In fact, it needs very little charge to retain its memory.
- ROM is used to store permanent, or semi-permanent data that persists even while the system is turned off.

SRAM vs DRAM

- There are many reasons to use an SRAM or a DRAM in a system design. Design tradeoffs include density, speed, volatility, cost, and features. All of these factors should be considered before you select a RAM for your system design.
  - **Speed.** The primary advantage of an SRAM over a DRAM is its speed. The fastest DRAMs on the market still require five to ten processor clock cycles to access the first bit of data
  - **Density.** Because of the way DRAM and SRAM memory cells are designed, readily available DRAMs have significantly higher densities than the largest SRAMs.

SRAM vs DRAM

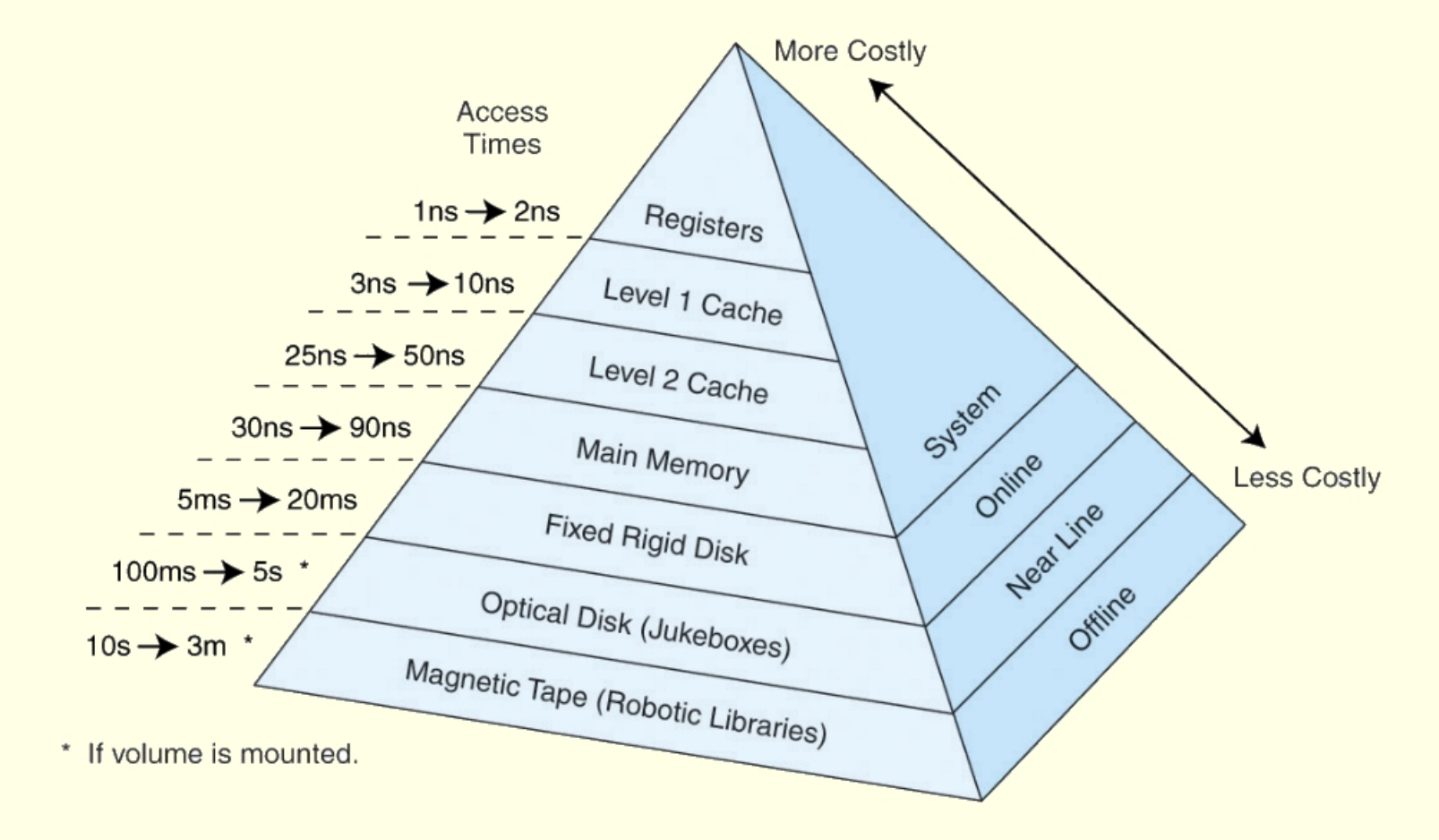
- **Volatility.** Unlike DRAMs, SRAM cells do not need to be refreshed. This means they are available for reading and writing data 100% of the time.
- **Cost.** If cost is the primary factor in a memory design, then DRAMs win hands down. If, on the other hand, performance is a critical factor, then a well-designed SRAM is an effective cost performance solution

6.3 The Memory Hierarchy

- Generally speaking, faster memory is more expensive than slower memory.
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.
- Small, fast storage elements are kept in the CPU, larger, slower main memory is accessed through the data bus.
- Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.

6.3 The Memory Hierarchy

- This storage organization can be thought of as a pyramid:



6.3 The Memory Hierarchy

- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, then main memory is queried. If the data is not in main memory, then the request goes to disk.
- Once the data is located, then the data, and a number of its **nearby data** elements are fetched into cache memory.

6.3 The Memory Hierarchy

- This leads us to some definitions.
  - A *hit* is when data is found at a given memory level.
  - A *miss* is when it is not found.
  - The *hit rate* is the percentage of time data is found at a given memory level.
  - The *miss rate* is the percentage of time it is not.
  - Miss rate = 1 - hit rate.
  - The *hit time* is the time required to access data at a given memory level.
  - The *miss penalty* is the time required to process a miss, including the time that it takes to replace a block of memory plus the time it takes to deliver the data to the processor.

6.3 The Memory Hierarchy

- An entire blocks of data is copied after a hit because the *principle of locality* tells us that once a byte is accessed, it is likely that a nearby data element will be needed soon.
- There are three forms of locality:
  - *Temporal locality*- Recently-accessed data elements tend to be accessed again.
  - *Spatial locality* - Accesses tend to cluster.
  - *Sequential locality* - Instructions tend to be accessed sequentially.

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.
- Unlike main memory, which is accessed by address, cache is typically accessed by content; hence, it is often called ***content addressable memory***.
- Because of this, a single large cache memory isn't always desirable-- it takes longer to search.

6.4 Cache Memory

- If the data has been copied to cache, the address of the data in cache is not the same as the main memory address.
- How does the CPU locate data when it has been copied into cache?
- **The CPU uses a specific mapping scheme that converts the main memory address into a cache location.**
- This address conversion is done by giving special significance to the bits in the main memory address. We divide the bits into distinct groups, called **fields**

6.4 Cache Memory

- The fields provide a **many-to-one** mapping between larger main memory and the smaller cache memory.
- Depending on the mapping scheme, we may have two or three fields.
- Many blocks of main memory map to a single block of cache. A *tag* field in the cache block distinguishes one cached memory block from another.

6.4 Cache Memory

- The simplest cache mapping scheme is *direct mapped cache*.
- In a direct mapped cache consisting of *N* blocks of cache, block *X* of main memory maps to cache block *Y* = *X* mod *N*.
- Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.
- Once a block of memory is copied into its slot in cache, a *valid* bit is set for the cache block to let the system know that the block contains valid data.

6.4 Cache Memory

- The diagram below is a schematic of what cache looks like.

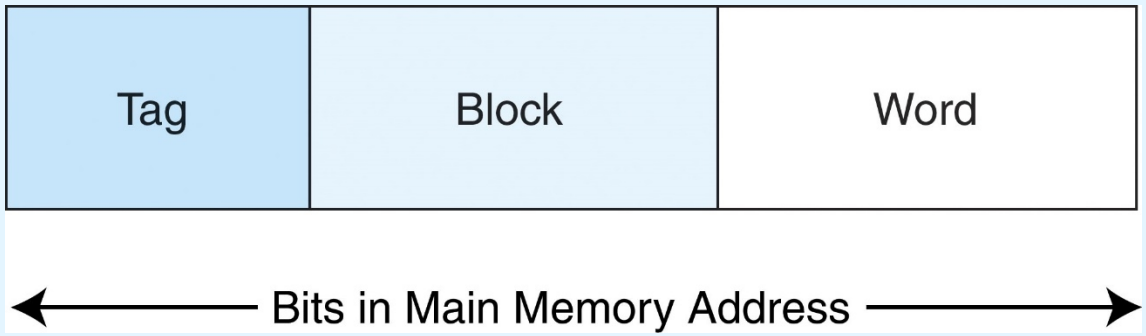
Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	----------		0
3	----------		0

- Block 0 contains multiple words from main memory, identified with the tag 00000000. Block 1 contains words identified with the tag 11110101.
- The other two blocks are not valid.

What could happen without having a valid bit?

6.4 Cache Memory

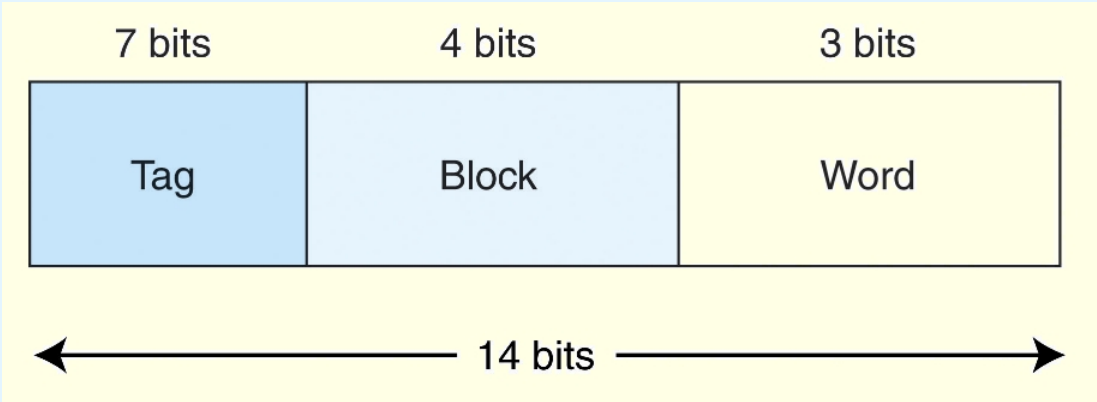
- To perform direct mapping, the binary main memory address is partitioned into the fields shown below



- The word filed(offset field) identifies a word from specific block.
- The block field select a unique block in cache
- A *tag* field in the cache block distinguishes one cached memory block from another.

6.4 Cache Memory

- The size of each field into which a memory address is divided **depends on the size of the cache**.
- Suppose our memory consists of 2^{14} words, cache has $16 = 2^4$ blocks, and each block holds 8 words.
  - Thus memory is divided into $2^{14} / 2^8 = 2^{11}$ blocks.
- For our field sizes, we know we need 4 bits for the block, 3 bits for the word, and the tag is what's left over:

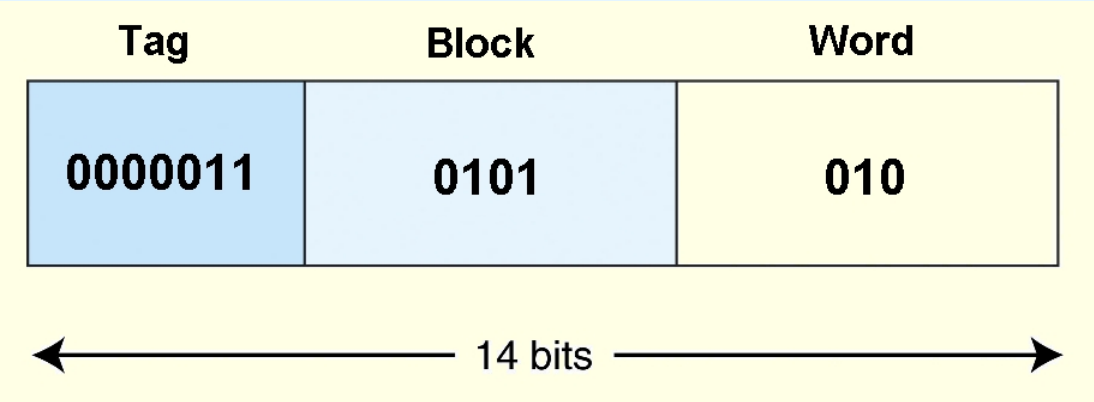


6.4 Cache Memory

- Question: Suppose a computer using direct mapped cache has 2^{20} words of main memory, and a cache of 32 blocks, where each cache block contains 16 words.
  - a. How many blocks of main memory are there?
  - b. What is the format of a memory address as seen by the cache, i.e., what are the sizes of the tag, block, and word fields?
  - c. To which cache block will the memory reference $0DB63_{16}$ map?

6.4 Cache Memory

- As an example, suppose a program generates the address **`1AA`**. In 14-bit binary, this number is: `00000110101010`.
- The first 7 bits of this address go in the tag field, the next 4 bits go in the block field, and the final 3 bits indicate the word within the block.



6.4 Cache Memory

- If subsequently the program generates the address **`1AB`**, it will find the data it is looking for in block `0101`, word `011`.

Tag	Block	Word
0000011	0101	010

- However, if the program generates the address, **`3AB`**, instead, the block loaded for address **`1AA`** would be evicted from the cache, and replaced by the blocks associated with the **`3AB`** reference.

6.4 Cache Memory

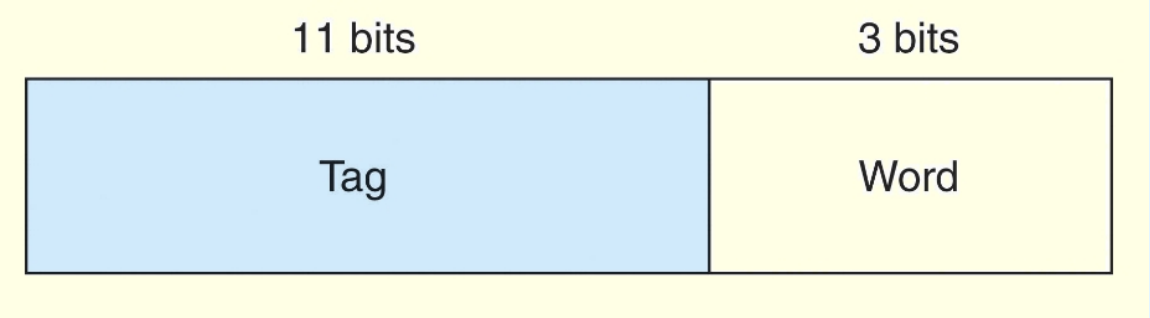
- Suppose a program generates a series of memory references such as: **`1AB, 3AB, 1AB, 3AB`**, . . . The cache will continually evict and replace blocks.
- The theoretical advantage offered by the cache is lost in this extreme case.
- This is the main disadvantage of direct mapped cache.
- Other cache mapping schemes are designed to prevent this kind of thrashing.

6.4 Cache Memory

- Instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go anywhere in cache.
- In this way, cache would have to fill up before any blocks are evicted.
- This is how *fully associative* cache works.
- A memory address is partitioned into only two fields: the tag and the word.

6.4 Cache Memory

- 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

- Question: Suppose a computer using fully associative cache has 2^{16} words of main memory, and a cache of 64 blocks, where each cache block contains 32 words.
  - a. How many blocks of main memory are there?
  - b. What is the format of a memory address as seen by the cache, i.e., what are the sizes of the tag and word fields?
  - c. To which cache block will the memory reference F8C9 map?

6.4 Cache Memory

- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus we must **devise an algorithm** to determine which block to evict from the cache.
- The block that is evicted is the *victim block*.
- There are a number of ways to pick a victim, we will discuss them shortly.

6.4 Cache Memory

- 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.

6.4 Cache Memory

- The number of cache blocks per set in set associative cache varies according to overall system design.
- For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
- Each set contains two different memory blocks.

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, . . .	1	----------		0
1	11110101	Words L, M, N, . . .	1	----------		0
2	----------		0	10111011	P, Q, R, . . .	1
3	----------		0	11111100	T, U, V, . . .	1

6.4 Cache Memory

- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and word, as shown below.
- As with direct-mapped cache, the word 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.

6.4 Cache Memory

- Suppose we have a main memory of 2^{14} bytes.
- This memory is mapped to a 2-way set associative cache having 16 blocks where each block contains 8 words.
- Since this is a 2-way cache, each set consists of 2 blocks, and there are 8 sets.
- Thus, we need 3 bits for the set, 3 bits for the word, giving 8 leftover bits for the tag:

6.4 Cache Memory

- Question: Assume a system's memory has 128M words. Blocks are 64 words in length and the cache consists of 32K blocks. Show the format for a main memory address assuming a 2-way set associative cache mapping scheme. Be sure to include the fields as well as their sizes.

6.4 Cache Memory

- Question: A 2-way set-associative cache consists of four sets. Main memory contains 2K blocks of eight words each.
  - a. Show the main memory address format that allows us to map addresses from main memory to cache. Be sure to include the fields as well as their sizes.
  - b. Compute the hit ratio for a program that loops 3 times from locations 8 to 51in main memory.

6.4 Cache Memory

- With fully associative and set associative cache, a *replacement policy* is invoked when it becomes necessary to evict a block from cache.
- An *optimal* replacement policy would be able **to look into the future** to see which blocks won't be needed for the longest period of time.
- Although it is impossible to implement an optimal replacement algorithm, it is instructive to use it as a benchmark for assessing the efficiency of any other scheme we come up with.

6.4 Cache Memory

- The replacement policy that we choose depends upon the locality that we are trying to optimize--usually, we are interested in **temporal locality**.
- A *least recently used* (LRU) algorithm keeps track of the last time that a block was assessed and evicts the block that has been unused for the longest period of time.
- The disadvantage of this approach is its complexity: LRU has to maintain an access history for each block, which ultimately slows down the cache.

6.4 Cache Memory

- *First-in, first-out* (**FIFO**) is a popular cache replacement policy.
- In FIFO, the block that has been in the cache the longest, regardless of when it was last used.
- A ***random* replacement** policy does what its name implies: It picks a block at random and replaces it with a new block.
- Random replacement can certainly evict a block that will be needed often or needed soon

6.4 Cache Memory

- The performance of hierarchical memory is measured by its *effective access time* (EAT).
- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.
- The EAT for a two-level memory is given by:

$$\text{EAT} = H \times \text{Access}_C + (1\text{-}H) \times \text{Access}_{MM}.$$

where H is the cache hit rate and Access_C and Access_{MM} are the access times for cache and main memory, respectively.

6.4 Cache Memory

- For example, consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.
- The EAT is:

  0.99(10ns) + 0.01(200ns) = 9.9ns + 2ns = 11ns.

- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

6.4 Cache Memory

- Cache replacement policies must also take into account ***dirty blocks***, those blocks that have been updated while they were in the cache.
- Dirty blocks must be written back to memory. A *write policy* determines how this will be done.
- There are two types of write policies, ***write through*** **and** ***write back***.
- Write through updates cache and main memory simultaneously on every write.

6.4 Cache Memory

- Write back (also called *copyback*) updates memory **only when** the block is selected for replacement.
- The disadvantage of write through is that memory must be updated with each cache write, which **slows down** the access time on updates.
- The advantage of write back is that memory traffic is minimized, but its disadvantage is that memory does not always agree with the value in cache, **causing problems in systems with many concurrent users**.

6.5 Virtual Memory

- Cache memory enhances performance by providing faster memory access speed.
- Virtual memory enhances performance by providing **greater memory capacity**, without the expense of adding main memory.
- Instead, a portion of a disk drive serves as an extension of main memory.
- The purpose of virtue memory is to use the hard disk as an extension of RAM

6.5 Virtual Memory

- In cache, a main memory address had to be transformed into a cache location, the same is true when using virtue memory
- Virtual address: the logical or program address that the **process uses**.
- *Physical address*: the actual memory address in physical memory.
- Mapping: the mechanism by which virtual address are translated into physical ones
- Page **frames**: the equal size chunks or blocks into which **main memory** is divided

6.5 Virtual Memory

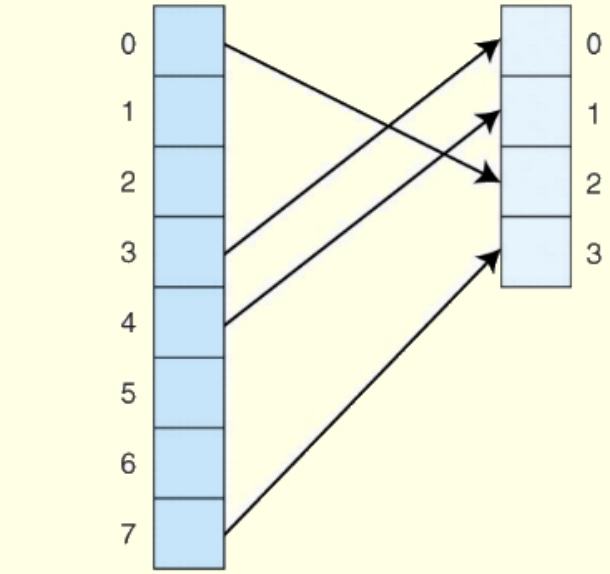
- **Pages**: The chunks or blocks into which **virtual memory** is divided, each equal in size to a page frame
- Paging: the process of **copying** a virtual page from disk to a page frame in main memory
- *Page faults: an event that* occur when a logical address requires that a page be brought in from disk.
- *Memory fragmentation* occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

6.5 Virtual Memory

- Main memory and virtual memory are divided into equal sized pages.
- The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- Further, the pages allocated to a process do not need to be stored contiguously-- either on disk or in memory.
- In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

6.5 Virtual Memory: Paging

- A page table keep track of where the various pages of the process reside (shown below).
- There is one page table for **each active process**.
- Typically **resides in main memory**



Page Table

Page	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

- When a process generates a virtual address, the **operating system** translates it into a physical memory address.
- To accomplish this, the **virtual address is divided into two fields: A *page* field, and an *offset* field**.
- The page field determines the page location of the address, and the offset indicates the location of the address within the page.
- The logical page number is translated into a physical page frame **through a lookup in the page table**.

6.5 Virtual Memory

- If **the valid bit** is zero in the page table entry for the logical address, this means that the page is not in memory and must be fetched from disk.
  - This is a **page fault**.
  - If necessary, a page is evicted from memory and is replaced by the page retrieved from disk, and the valid bit is set to 1.
- If the valid bit is 1, the virtual page number is replaced by the physical frame number.
- The data is then accessed by adding the offset to the physical frame number.

6.5 Virtual Memory: summarize paging steps

1. Extract the page number from the virtual address.

2. Extract the offset from the virtual address.

3. Translate the page number into a physical page frame number by accessing the page table.

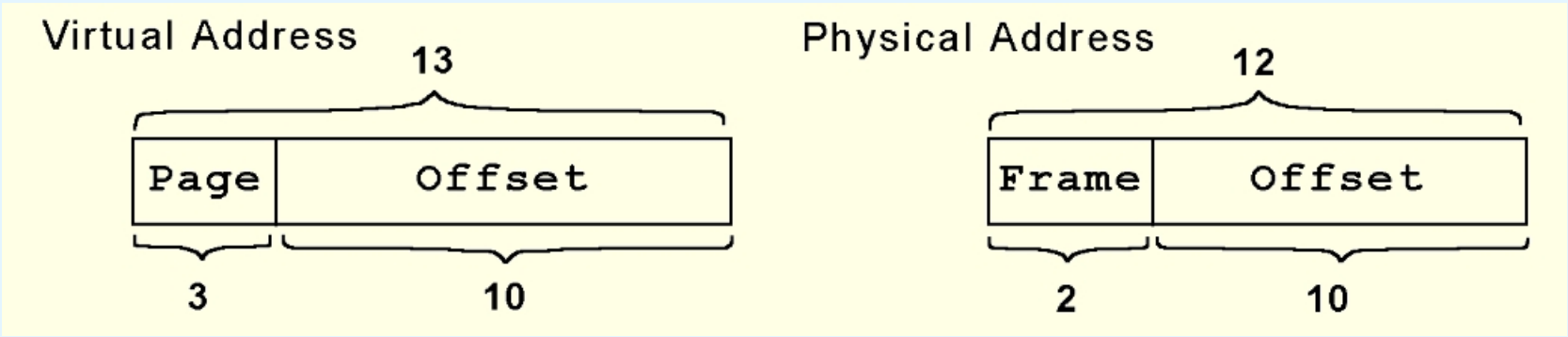
- **A.** Look up the page number in the page table (using the virtual page number as an index).
- **B.** Check the valid bit for that page.
- **1. If the valid bit = 1, the page is in memory.**
  - **a. Replace the virtual page number with the actual frame number.**
  - **b. Access data at offset in physical page frame by adding the offset to the frame number for the given virtual page.**

6.5 Virtual Memory: summarize paging steps

- **2. If the valid bit = 0, the system generates a page fault and the operating system must intervene to**
  - **a.** Locate the desired page on disk.
  - **b.** Find a free page frame (this may necessitate removing a "victim" page from memory and copying it back to disk if memory is full).
  - **c.** Copy the desired page into the free page frame in main memory.
  - **d.** Update the page table. (The virtual page just brought in must have its frame number and valid bit in the page table modified. If there was a "victim" page, its valid bit must be set to zero.)
  - **e.** Resume execution of the process causing the page fault, continuing to Step B1.

6.5 Virtual Memory

- As an example, suppose a system has a virtual address space of 8K and a physical address space of 4K, 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.



6.5 Virtual Memory

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

Page Table

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

	Addresses
Page 0 :	0 - 1023
1 :	1024 - 2047
2 :	2048 - 3071
3 :	3072 - 4095
4 :	4096 - 5119
5 :	5120 - 6143
6 :	6144 - 7167
7 :	7168 - 8191

6.5 Virtual Memory

- The address 1010101010011_2 is converted to physical address 010101010011 because the page field 101 is replaced by frame number 01 through a lookup in the page table.

Page Table

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

	Addresses
Page 0 :	0 - 1023
1 :	1024 - 2047
2 :	2048 - 3071
3 :	3072 - 4095
4 :	4096 - 5119
5 :	5120 - 6143
6 :	6144 - 7167
7 :	7168 - 8191

6.5 Virtual Memory

- What happens when the CPU generates address 1000000000100_2?

Page Table

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

	Addresses
Page 0 :	0 - 1023
1 :	1024 - 2047
2 :	2048 - 3071
3 :	3072 - 4095
4 :	4096 - 5119
5 :	5120 - 6143
6 :	6144 - 7167
7 :	7168 - 8191

6.5 Virtual Memory

- Selecting an appropriate page size is very difficult
- The larger the page size, the smaller the page table, but worse internal fragmentation
- Larger page sizes mean fewer actual transfers , but waste resources by transferring data that may not be necessary.

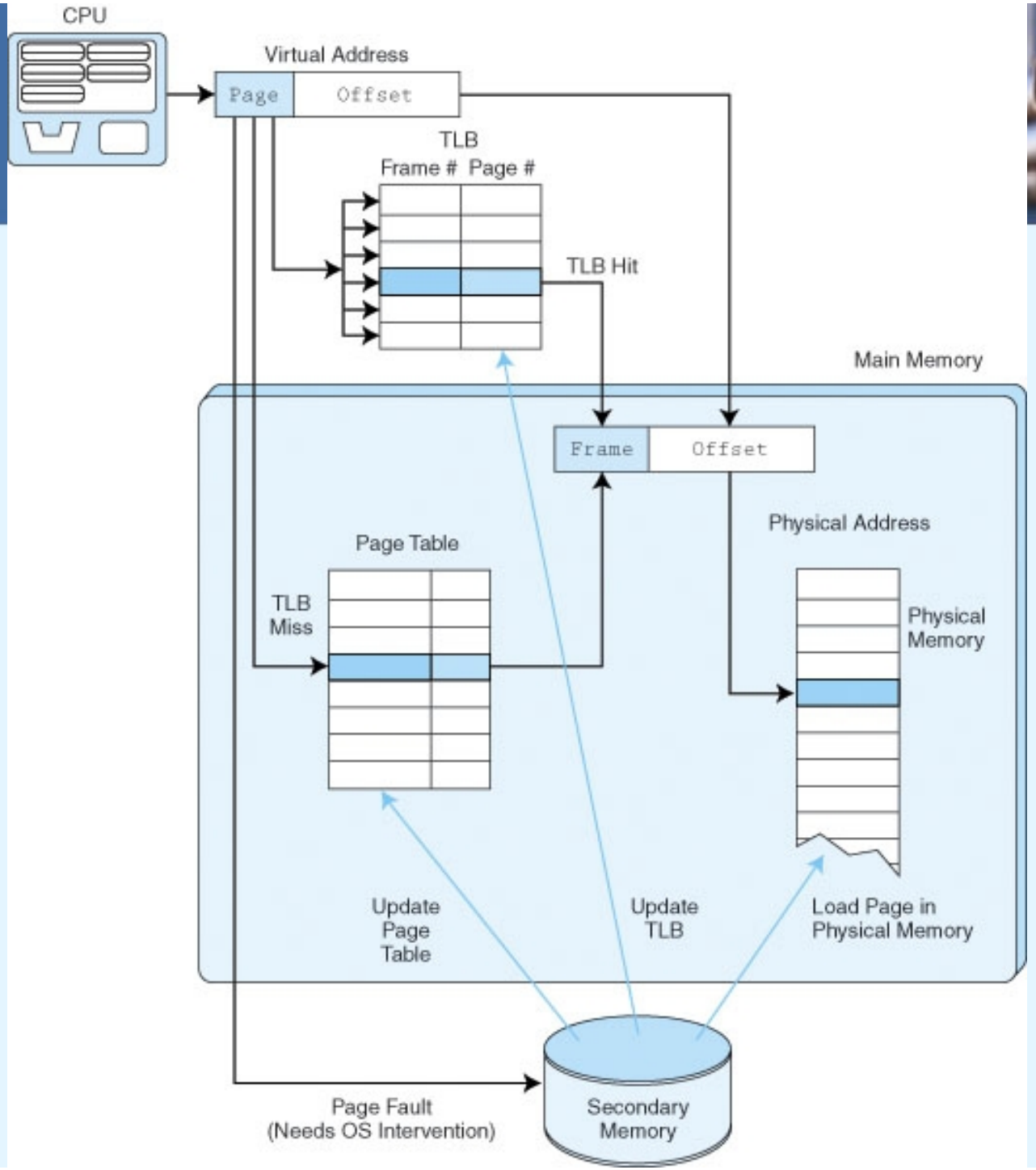
6.5 Virtual Memory

- We said earlier that 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, 396ns.

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).
- TLB: storing the **most recent** page lookup values in a page table cache, implemented as associative cache


CPU
Virtual Address
Page
Offset
TLB
Frame # Page #
TLB Hit
Main Memory
Frame
Offset
Physical Address
Page Table
TLB
Miss
Physical
Memory
Update
Page
Table
Update
TLB
Load Page in
Physical Memory
Page Fault
(Needs OS Intervention)
Secondary
Memory

6.5 Virtual Memory

1. Extract the page number from the virtual address.

2. Extract the offset from the virtual address.

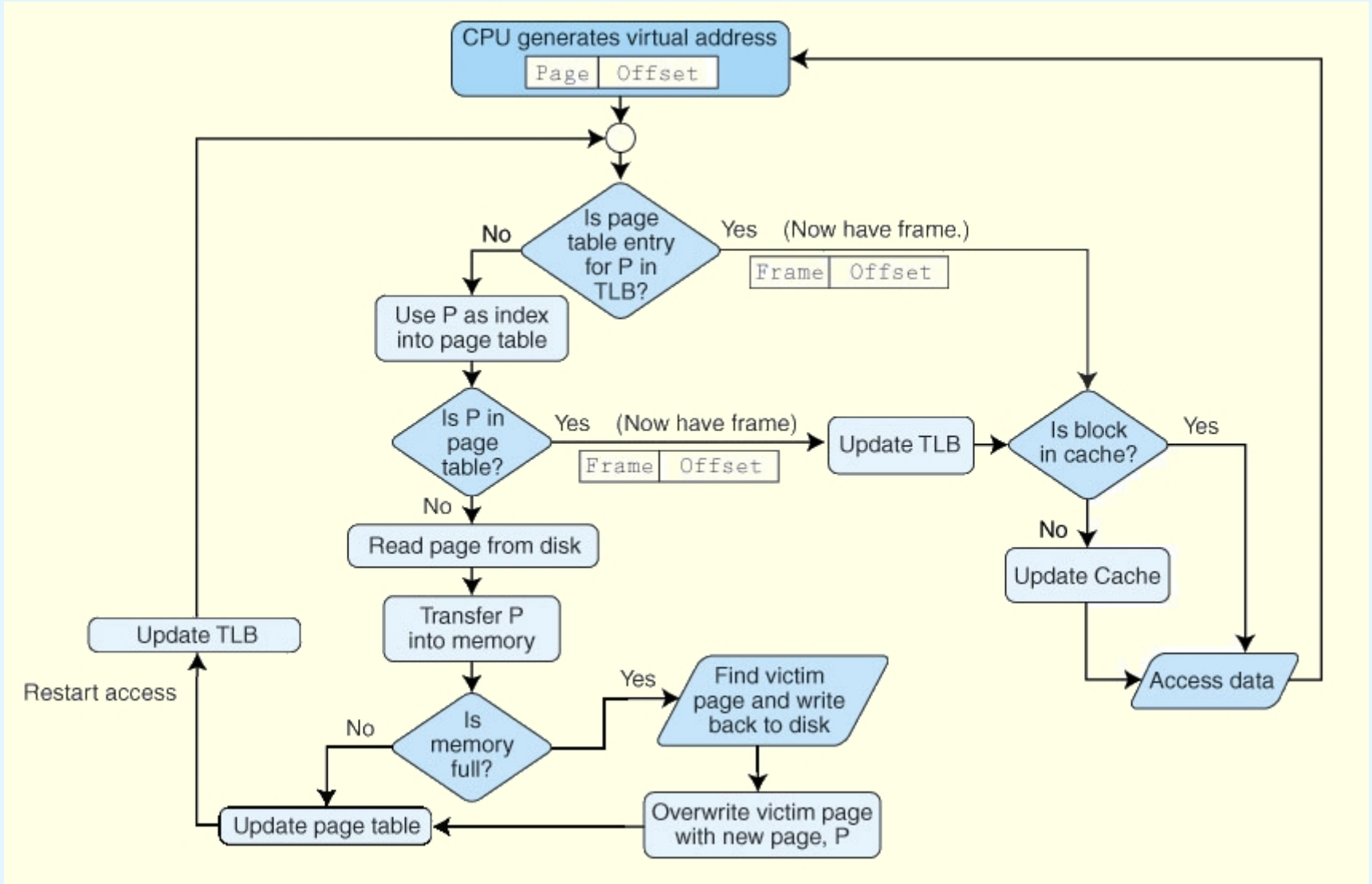
3. Search for the virtual page number in the TLB.

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.

6.5 Virtual Memory



6.5 Virtual Memory

- Question: You have a virtual memory system with a two-entry TLB, a 2-way set associative cache and a page table for a process P. Assume cache blocks of 8 words and page size of 16 words. In the system below, main memory is divided up into blocks, where each block is represented by a letter. Two blocks equals one frame.

Given the system state as depicted above, answer the following questions

0	3
4	1

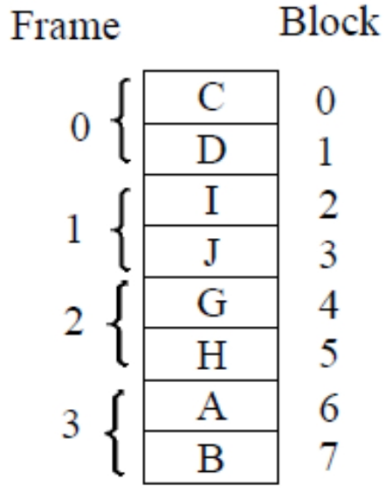
TLB

Set 0	tag	C	tag	I
Set 1	tag	D	tag	H

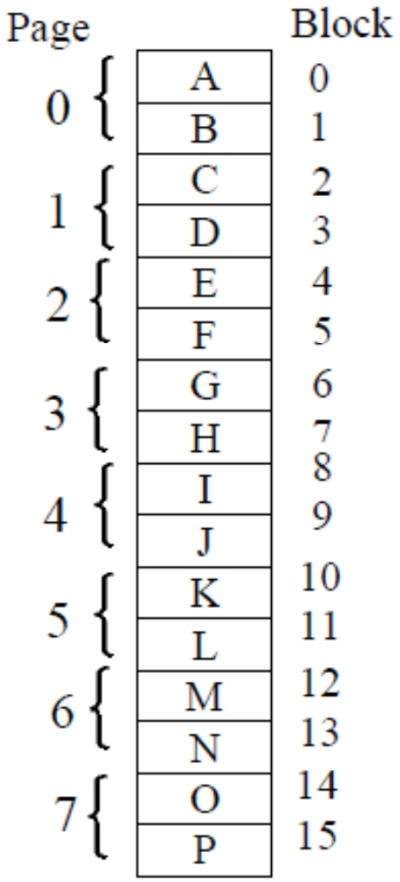
Cache

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

Page Table



Main Memory



Virtual Memory
For Process P

6.5 Virtual Memory

- a. How many bits are in a virtual address for process P? Explain.
- b. How many bits are in a physical address? Explain.
- c. Show the address format for virtual address 18_{10} (specify field name and size) that would be used by the system to translate to a physical address and then translate this virtual address into the corresponding physical address

6.5 Virtual Memory

- d. Show the format for a physical address (specify the field names and sizes) that is used to determine the cache location for this address. Explain how to use this format to determine where physical address 54 would be located in cache.
- e. Given virtual address 25_{10} is located on virtual page 1, offset 9. Indicate exactly how this address would be translated to its corresponding physical address and how the data would be accessed. Include in your explanation how the TLB, Page Table, Cache and Memory are used.

6.5 Virtual Memory

- Advantages
  - Programs are no longer restricted by the amount of physical memory that is available
  - makes it much easier to write programs because the programmer no longer has to worry about the physical address space limitations
  - virtual memory also permits us to run more program at the same time

6.5 Virtual Memory

- disadvantages
  - adds an extra memory reference when accessing data
  - extra resource consumption(the memory overhead for storing page tables), page tables may take up a significant portion of physical memory
  - Virtual memory and paging also require special hardware(MMU) and operating system support

6.5 Virtual Memory

- Another approach to virtual memory is the use of *segmentation*.
- 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.

6.5 Virtual Memory

- Both paging and segmentation can cause fragmentation.
- Paging is subject to ***internal*** fragmentation because a process may not need the entire range of addresses contained within the page. Thus, there may be many pages containing unused fragments of memory.
- Segmentation is subject to ***external*** fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over time.

6.5 Virtual Memory

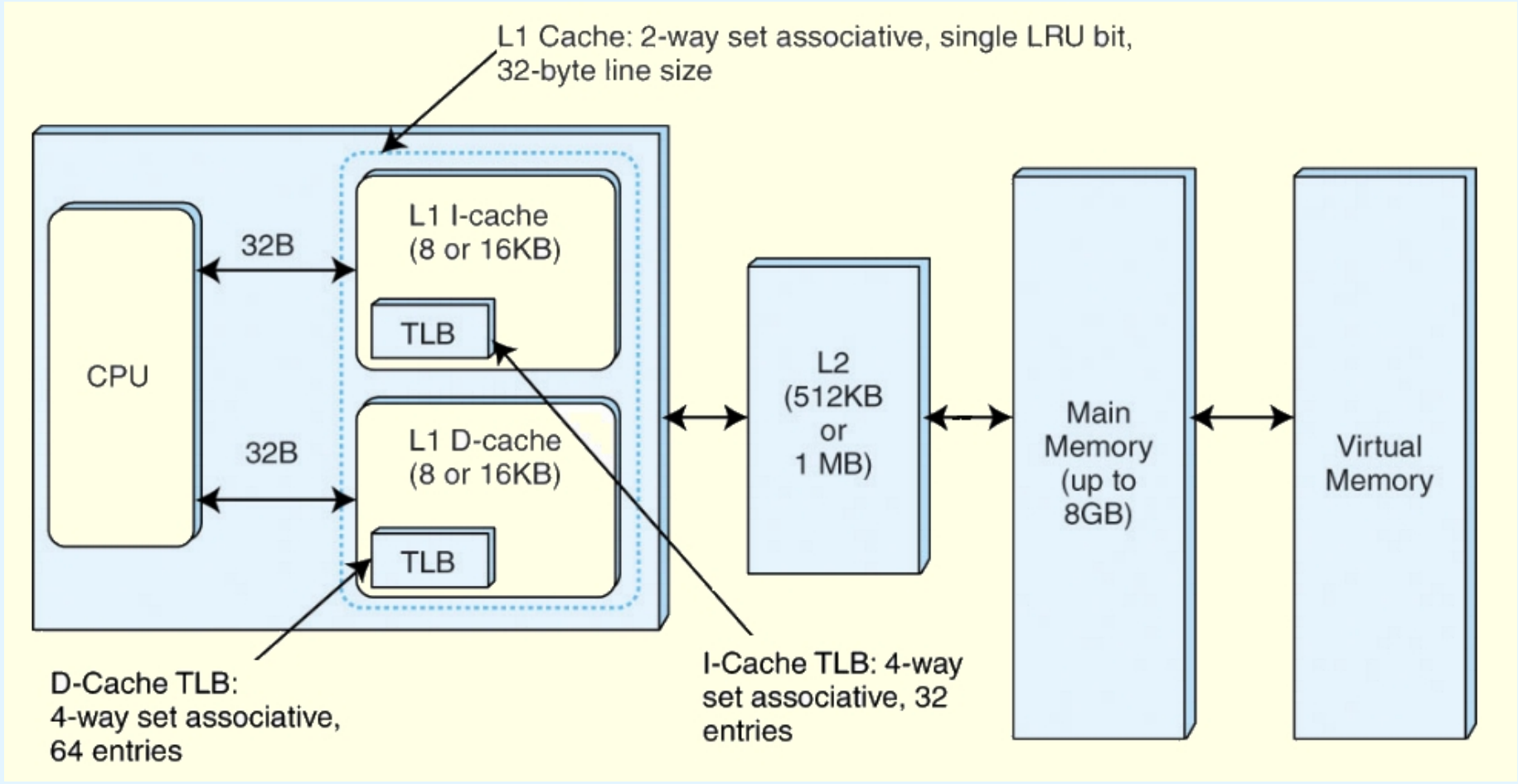
- Large page tables are cumbersome and slow, but with its uniform memory mapping, page operations are fast. Segmentation allows fast access to the segment table, but segment loading is labor-intensive.
- Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.
- Each segment has a page table. This means that a memory address will have three fields, one for the segment, another for the page, and a third for the offset.

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



Chapter 6 Conclusion

- Computer memory is organized in a hierarchy, with the smallest, fastest memory at the top and the largest, slowest memory at the bottom.
- Cache memory gives faster access to main memory, while virtual memory uses disk storage to give the illusion of having a large main memory.
- Cache maps blocks of main memory to blocks of cache memory. Virtual memory maps page frames to virtual pages.
- There are three general types of cache: Direct mapped, fully associative and set associative.

Chapter 6 Conclusion

- With fully associative and set associative cache, as well as with virtual memory, replacement policies must be established.
- Replacement policies include LRU, FIFO, or LFU. These policies must also take into account what to do with dirty blocks.
- All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.

Chapter 6 Homework

- P269: 9, 11
- P272：16