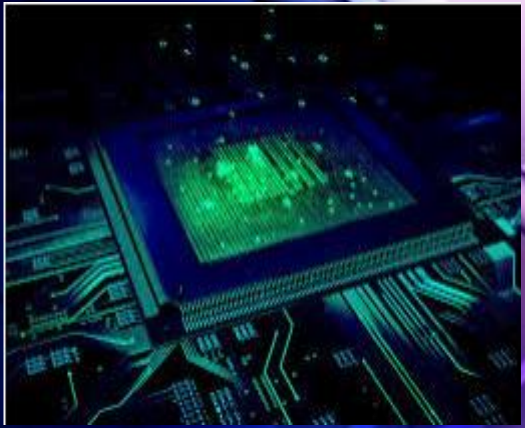
El-Shorouk Academy

The Higher Institute of Engineering, El-Shorouk City

Communications and Computer Engineering Department



Computer Architecture

CCE 394 [2013],
CCE 331 [2019]

TERM 2

Dr. Ahmed El-Shafei

Computer Architecture (2)
بنية الحاسب (٢)

Lecture Twelve

Week 1 : 15/2/2023 **2013**

Week 1 : 15/2/2023 **2019**

Computer Architecture (2)

اسم المقرر	**Computer Architecture (2)** **بنية الحاسب (٢)**
رقم المقرر	**CCE 394 [2013], CCE 331 [2019]** **الفرقة الثالثة هندسة الحاسبات و التحكم** ***Third Year Computer and Control Engineering***
توزيع الدرجات	***Grading System (next Slide)***
عدد الساعات	محاضرة : ٢ ٢ ساعة أسبوعيا فصل : ١ ٢ ساعة أسبوعيا معمل: ١ ٢ ساعة أسبوعيا الإجمالى : ٤ ٦ ساعات أسبوعيا

Instructor : Dr. Ahmed El-Shafei, email: dr.ahmed.alshafaay@sha.edu.eg
Graduate Teaching Assistant : *Eng. Aya Mostafa,* email: Aya.mostafa@sha.edu.eg
Eng. Elshefaa. Atea, email: elsfefaa.atea@sha.edu.eg

Computer Architecture (2)

بنية الحاسب (٢)

Grading System

Final written Exam :		*40%* *60%*	60 60
Teacher opinion ………………………………………………………………			45 20
Attendance			-
Assignments and reports		*30%* *20%*	-
Quiz	**40% of 45 20 marks**		**18 8**
Mid-term exam	**60% of 45 20 marks**		**27 12**
Practical / Oral ………………………………………………………………			45 20
practical attendance	**10% of 45 20 marks**		**4 2**
Lab Reports	**10% of 45 20 marks**	*30%* *20%*	**5 2**
Final / Practical Exam	**80% of 45 20 marks**		**36 16**
Total		*100%*	150 100

Computer Architecture (2)
بنية الحاسب (٢)

Course Description:

- The structure and behavior of digital computers on several levels, functional system for computer components, sorting computer orders and synchronization, data transfer, the representation of numbers, remember formation, operating codes in the computer, combination orders, titles patterns computer, financing elements, memory organization, computer registers, and the organization of the microprocessor, arithmetic and logic operations

- هيكل وسلوك الحاسبات الرقمية علي عدة مستويات،المنظومة الوظيفية لمكونات الحاسب، ترتيب أوامر الحاسب والتزامن، نقل البيانات، تمثيل الارقام، تكوين البيانات أكواد التشغيل في الحاسب، مجموعة الاوامر، أنماط العناوين بالحاسب ،عناصر التمويل، تنظيم الذاكرة، مسجلات الحاسب، تنظيم المعالج الدقيق، العمليات الحسابية والمنطقية

- Prerequisite: **CCE 391**

Computer Architecture (1)
بنية الحاسب (١)

- ***Essential book:***
  - **Linda Null and Julia Lobur, "The Essentials of Computer Organization and Architecture", fifth edition, Jones & Bartlett Pub, 2018.**
    ISBN-13: 978-1284123036, ISBN-10: 1284123030
- ***Recommended books:***
  - **Patterson, Hennessy, David A. Patterson, "Computer Architecture: A quantitative Approach", sixth edition, Morgan Kaufmann, 2019.**
  - **Joseph D. Dumas II, "Computer Architecture: Fundamentals and principles of Computer Design", second edition, Published December 1, 2017.**
  - **David A. Patterson, John L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", fifth edition, Morgan Kaufmann, 2014.**
  - **M. Morris Mano, "Computer System Architecture", Third edition, Pearson 1993.**

Computer Architecture (2)
بنية الحاسب (٢)

Course Learning Objectives:

The main objectives of this course is to:

- **Define the concept of architecture and incorporate parameters to evaluate and analyze the performance**
- **Explain the impact of the ISA (*instruction set architecture*) on the architecture and performance, understanding the design principles of the ISA**
- **Identify the pipelining as a basic technique for increasing CPU performance as well as design, planning and control of pipeline units**
- **Understanding the evolution of the architectures and the differences between CISC (*Complex Instruction Set Computer*) and RISC (*Reduced Instruction Set Computer*) approaches**
- **Explain techniques for improving the performance of memory and input/output system**
- **Recognize the limitations of classical architectures and the importance of parallelism**
- **Know and use the usual terminology and the language of the subject and employ it correctly both orally and in writing**

Computer Architecture (2)
بنية الحاسب (٢)

Course Contents:

1. Computer performance measurement methods, criteria and pitfalls.
2. Principles of instruction set design, the role of compiler and optimizations, historical perspective on instruction set design.
3. Principle of pipeline processor, hazards and design considerations.
4. Exception handling and long latency operations in pipeline design.
5. Role of compiler and speculative execution on performance.
6. Instruction level parallelism and processor design.
7. Memory technology.
8. Cache design and its impact on performance.
9. Memory hierarchy, virtual memory and their cost/performance.
10. I/O systems.

Reports Formats:

- **All reports in PDF format unless otherwise stated.**
- **File name is the student *ID_xx*, *where xx : report is number***
- **The cover page should include:**
  - **Student ID,**
  - **Student Name,**
  - **Section number,**
  - **Student Serial,**
  - **Report title, and**
  - **report date**

Chapter 6

Memory

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.

- 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 (1 of 2)

- 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).

- DRAM 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 (2 of 2)

- **SRAM consists of circuits similar to the D flip-flop that we studied in <u>Chapter 3.</u>**

- **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 <u>permanent</u>, or <u>semi-permanent</u> data that persists even while the system is turned off.**

6.3 The Memory Hierarchy (1 of 6)

- **Generally speaking, faster memory is more expensive than slower memory.**

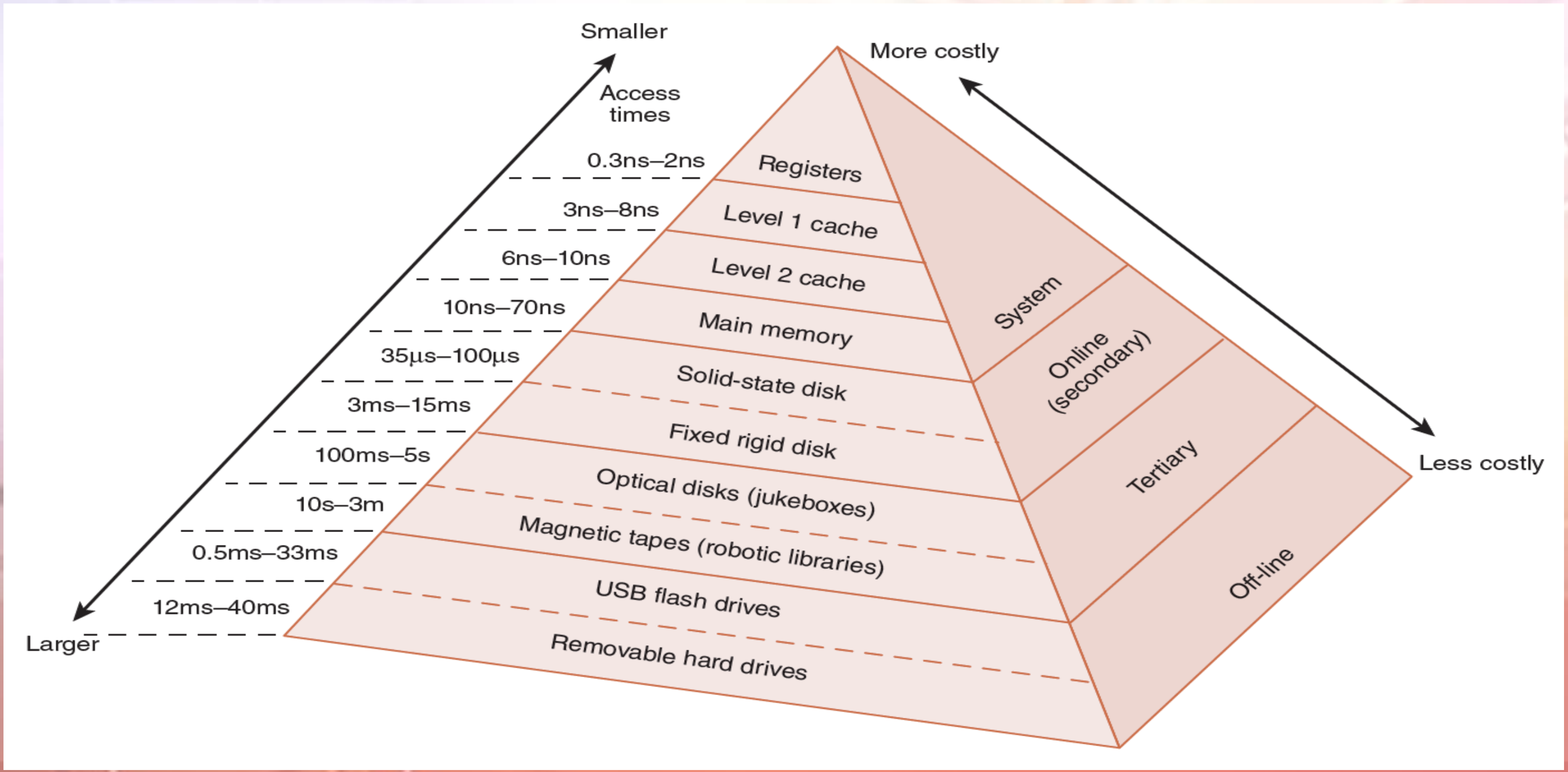
- **To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.**

- **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 (2 of 6)

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



6.3 The Memory Hierarchy (3 of 6)

- **We are most interested in the memory hierarchy that involves registers, cache, main memory, and virtual memory.**
- **Registers are storage locations available on the processor itself.**
- **Virtual memory is typically implemented using a hard drive; it extends the address space from RAM to the hard drive.**
- **Virtual memory provides more space: Cache memory provides speed.**

6.3 The Memory Hierarchy (4 of 6)

- 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 (5 of 6)

- **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 (6 of 6)

- An entire block of data is copied after a hit because the ***principle of locality*** tells us that once a byte is accessed, it's 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.**

Week 2 : 22/2/2023 2013

Week 2 : 22/2/2023 2019

6.4 Cache Memory ()

- ❑ **When cache hit occurs,**
  - ➢ **The required word is present in the cache memory.**
  - ➢ **The required word is delivered to the CPU from the cache memory.**

- ❑ **When cache miss occurs,**
  - ➢ **The required word is not present in the cache memory.**
  - ➢ **The page containing the required word must be mapped from the main memory.**
  - ➢ **This mapping is performed using cache mapping techniques.**

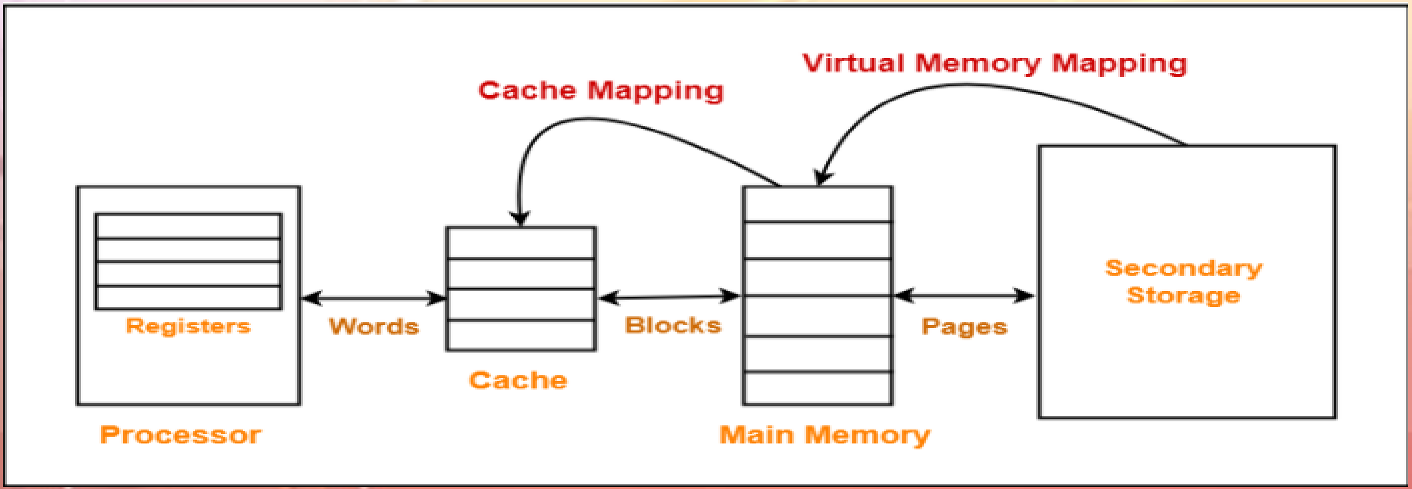
6.4 Cache Memory ()

❑ **Cache Mapping-**

- Cache mapping defines how a block from the main memory is mapped to the cache memory in case of a cache miss.

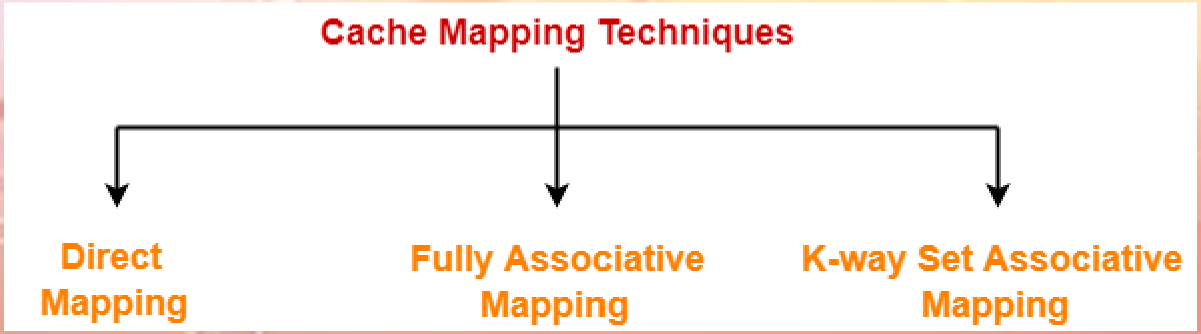
OR

- Cache mapping is a technique by which the contents of main memory are brought into the cache memory.
- The following diagram illustrates the mapping process-



6.4 Cache Memory ()

- Now, before proceeding further, it is important to note the following points
  - Main memory is divided into equal size partitions called as blocks or frames.
  - Cache memory is divided into partitions having same size as that of blocks called as lines.
  - During cache mapping, block of main memory is simply copied to the cache and the block is not actually brought from the main memory.
- **Cache Mapping Techniques-**
  - Cache mapping is performed using following three different techniques-



6.4 Cache Memory ()

1. **Direct Mapping-**

In direct mapping,

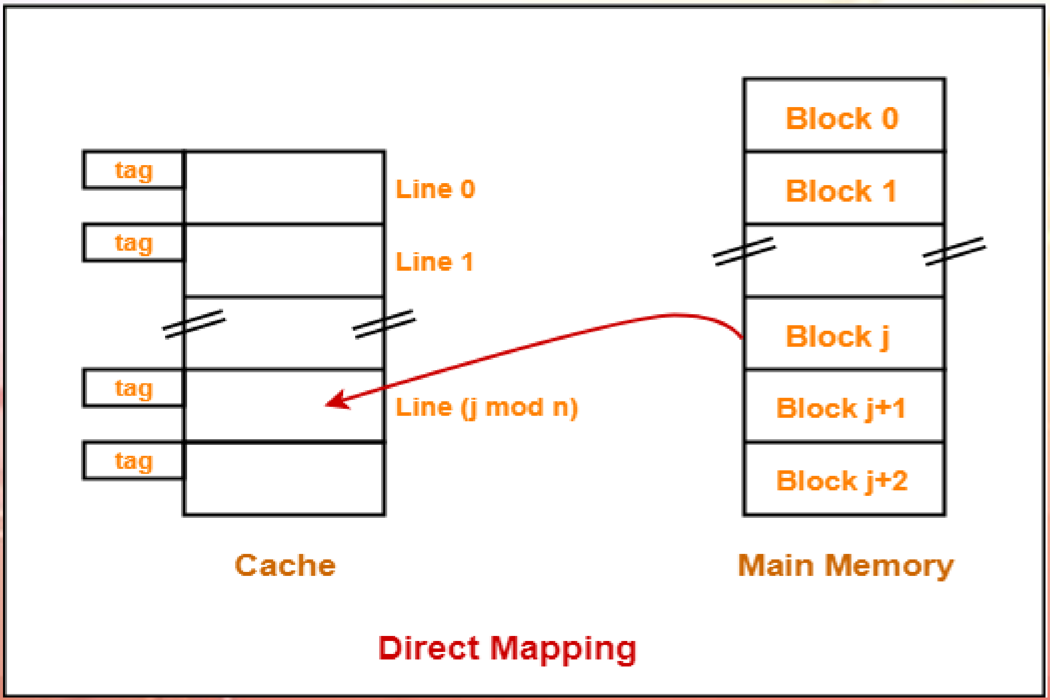
- **A particular block of main memory can map only to a particular line of the cache.**
- **The line number of cache to which a particular block can map is given by-**

Cache line number
= (Main Memory Block Address) Modulo (Number of lines in Cache)

6.4 Cache Memory ()

Example-

- Consider cache memory is divided into ‘n’ number of lines.
- Then, block ‘j’ of main memory can map to line number (j mod n) only of the cache.



6.4 Cache Memory ()

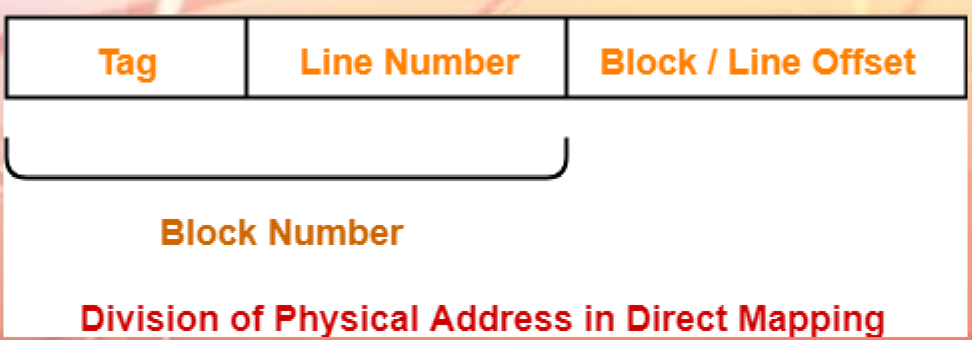
- <u>**Need of Replacement Algorithm-**</u>

  In direct mapping,

  - **There is no need of any replacement algorithm.**
  - **This is because a main memory block can map only to a particular line of the cache.**
  - **Thus, the new incoming block will always replace the existing block (if any) in that line.**

- <u>**Division of Physical Address-**</u>

  - **In direct mapping, the physical address is divided as-**



Division of Physical Address in Direct Mapping

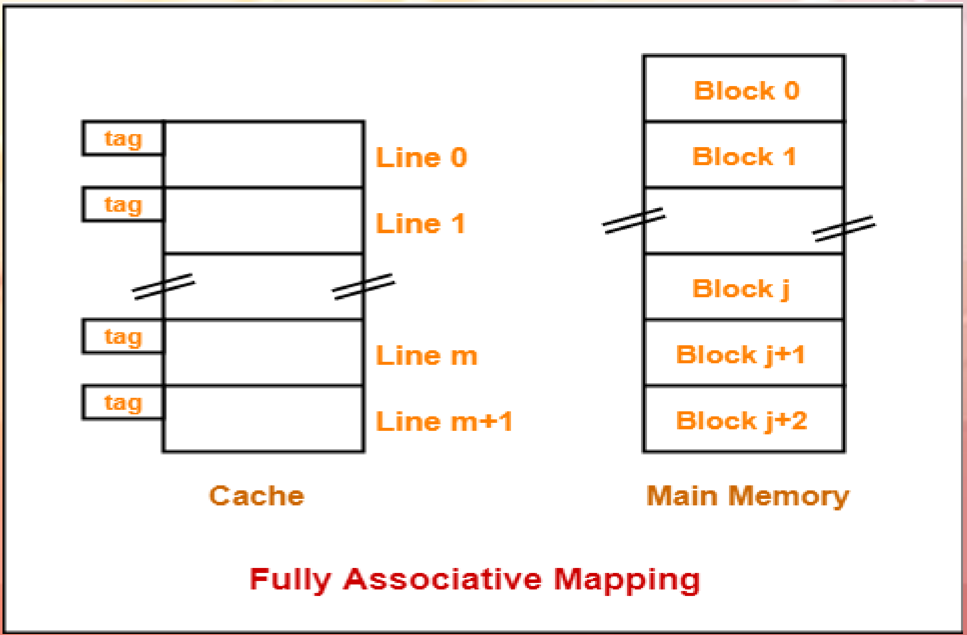
6.4 Cache Memory ()

2. Fully Associative Mapping-

- In fully associative mapping,
  - ➢ A block of main memory can map to any line of the cache that is freely available at that moment.
  - ➢ This makes fully associative mapping more flexible than direct mapping.

Example-

Consider the following scenario-



Here,
- ➢ All the lines of cache are freely available.
- ➢ Thus, any block of main memory can map to any line of the cache.
- ➢ Had all the cache lines been occupied, then one of the existing blocks will have to be replaced.

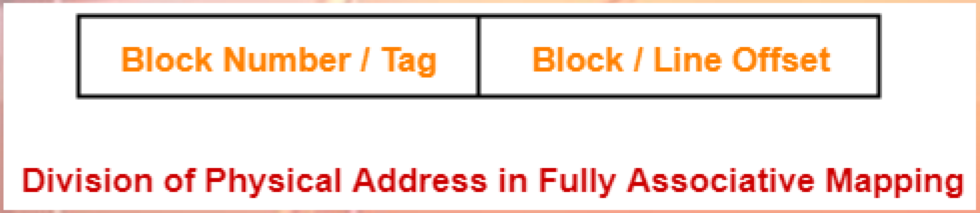
6.4 Cache Memory ()

- **Need of Replacement Algorithm-**

  In fully associative mapping,
  - **A replacement algorithm is required.**
  - **Replacement algorithm suggests the block to be replaced if all the cache lines are occupied.**
  - **Thus, replacement algorithm like FCFS Algorithm, LRU Algorithm etc is employed.**

- **Division of Physical Address-**

  In fully associative mapping, the physical address is divided as-



Division of Physical Address in Fully Associative Mapping

6.4 Cache Memory ()

3. K-way Set Associative Mapping-

In k-way set associative mapping,

- **Cache lines are grouped into sets where each set contains k number of lines.**
- **A particular block of main memory can map to only one set of the cache.**
- **However, within that set, the memory block can map any cache line that is freely available.**
- **The set of the cache to which a particular block of the main memory can map is given by-**

**Cache set number
= (Main Memory Block Address) Modulo (Number of sets in Cache)**

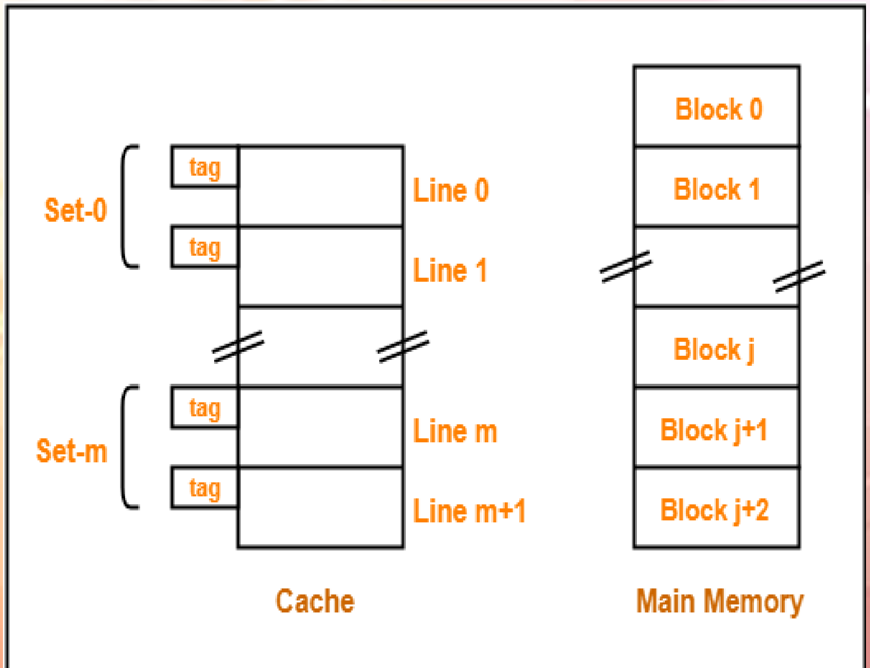
6.4 Cache Memory ()

Example-

Consider the following example of 2-way set associative mapping-

Here,

- k = 2 suggests that each set contains two cache lines.
- Since cache contains 6 lines, so number of sets in the cache = 6 / 2 = 3 sets.
- Block 'j' of main memory can map to set number (j mod 3) only of the cache.
- Within that set, block 'j' can map to any cache line that is freely available at that moment.
- If all the cache lines are occupied, then one of the existing blocks will have to be replaced.



2-Way Set Associative Mapping

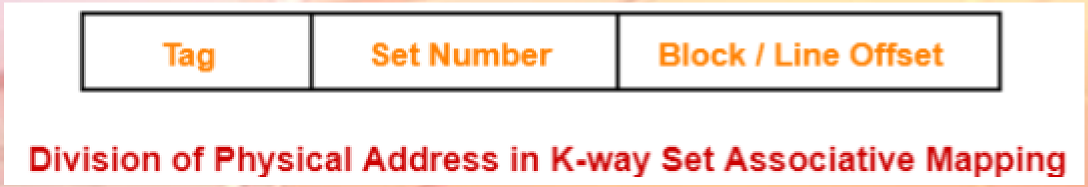
6.4 Cache Memory ()

Need of Replacement Algorithm-

- **Set associative mapping is a combination of direct mapping and fully associative mapping.**
- **It uses fully associative mapping within each set.**
- **Thus, set associative mapping requires a replacement algorithm.**

Division of Physical Address-

In set associative mapping, the physical address is divided as-



Tag	Set Number	Block / Line Offset

Division of Physical Address in K-way Set Associative Mapping

6.4 Cache Memory ()

<u>Special Cases-</u>

- If k = 1, then k-way set associative mapping becomes direct mapping i.e.

  1-way Set Associative Mapping ≡ Direct Mapping

- If k = Total number of lines in the cache, then k-way set associative mapping becomes fully associative mapping.

6.4 Cache Memory (1 of 45)

- **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**

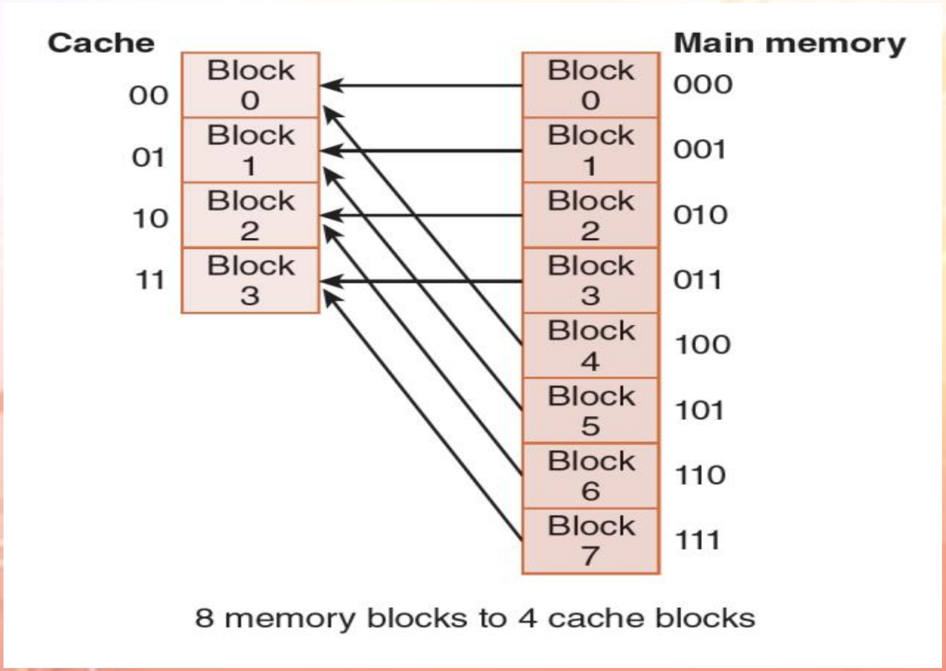
6.4 Cache Memory (2 of 45)

- **CPU generates main memory Address which is mapped by a Mapping Scheme to a Cache Location.**
- **The following are Mapping schemes:**
  - **Direct mapping,**
  - **fully associative mapping,**
  - **and set-associative mapping.**
- **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** **$Y = X \bmod N$**
- **Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory (7 mod 10, 17 mod 10, 27 mod 10, 37 mod 10)**

The next slide illustrates this mapping concept.

6.4 Cache Memory (3 of 45)

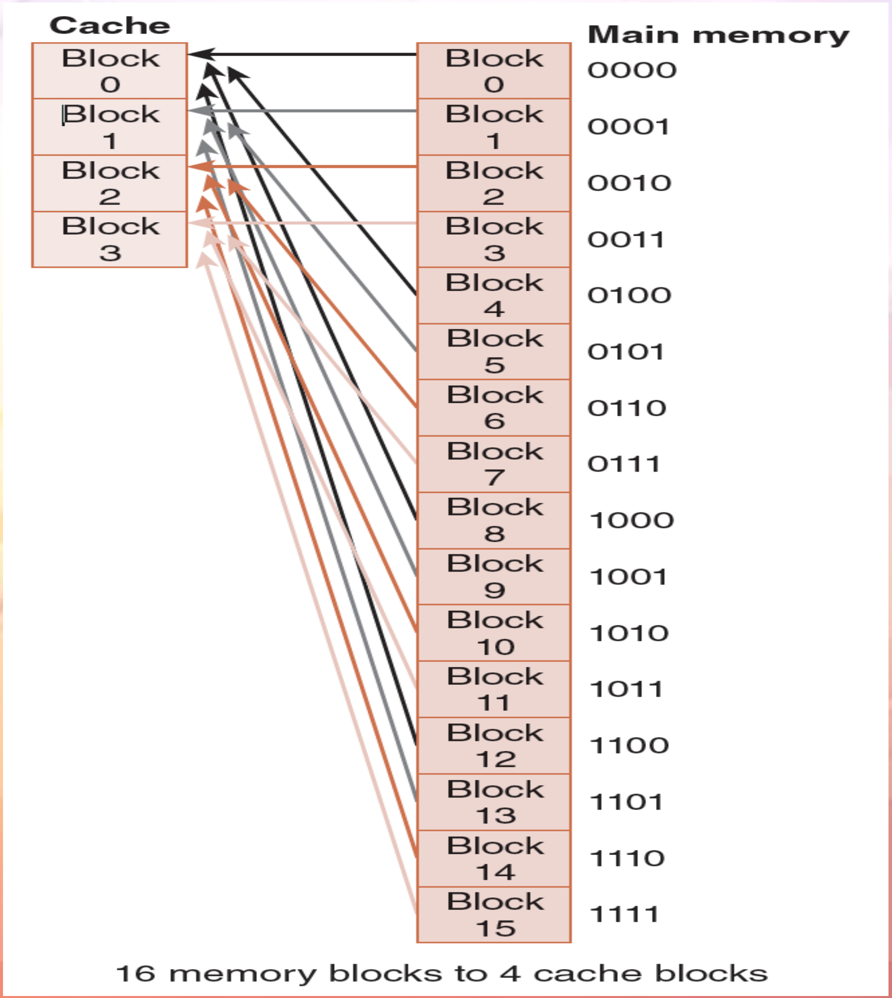
- **With direct mapped cache consisting of 4 blocks of cache, block *X* of main memory maps to cache block *Y* = *X* mod 4.**



8 memory blocks to 4 cache blocks

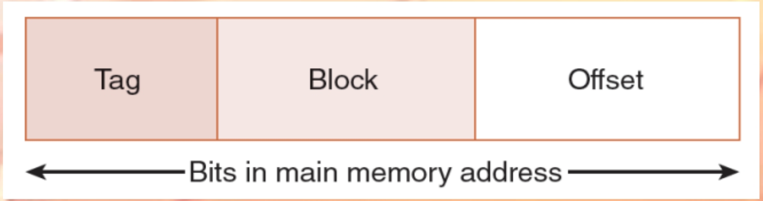
6.4 Cache Memory (4 of 45)

- A larger example.



6.4 Cache Memory (5 of 45)

- 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.



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

6.4 Cache Memory (6 of 45)

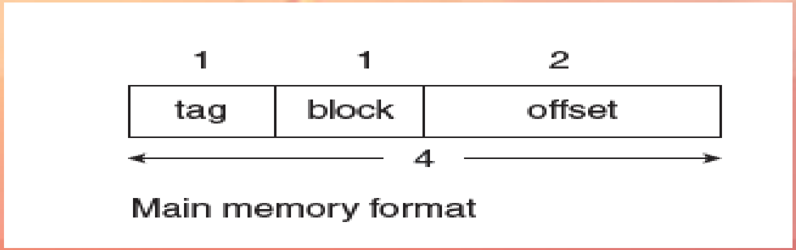
- Example 6.1: Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.

- This means Block 0 and 2 of main memory map to Block 0 of cache and Blocks 1 and 3 of main memory map to Block 1 of cache.

- Using the tag, block, and offset fields, we can see how main memory maps to cache as follows.

6.4 Cache Memory (7 of 45)

- Example 6.1: Cont’d. Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.
  - First, we need to determine the address format for mapping. Each block is 4 bytes, so the offset field must contain 2 bits; there are 2 blocks in cache, so the block field must contain 1 bit; this leaves 1 bit for the tag (as a main memory address has 4 bits because there are a total of $2^4 = 16$ bytes).

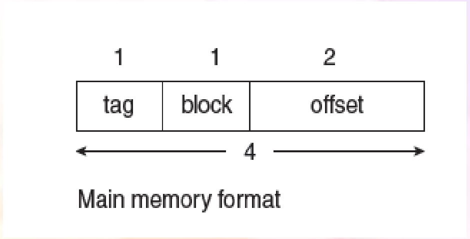


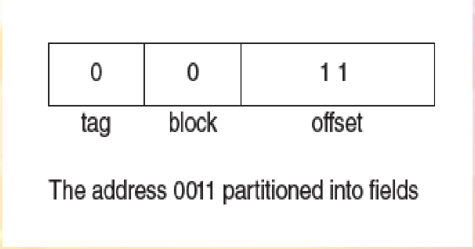
Computer Architecture (2)

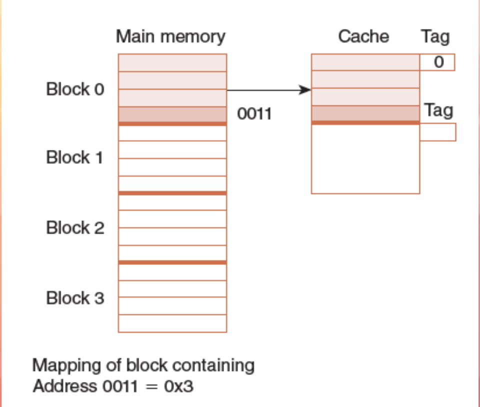
بنية الحاسب (٢)

6.4 Cache Memory (8 of 45)

- **Example 6.1: Cont'd.**
  - Suppose we need to access main memory address 3_{16} (0x0011 in binary). If we partition 0x0011 using the address format from Figure a, we get Figure b.
  - Thus, the main memory address 0x0011 maps to cache block 0.
  - Figure c shows this mapping, along with the tag that is also stored with the data.

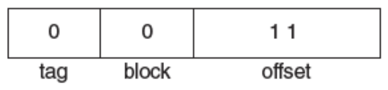




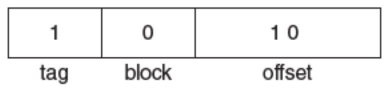


The next slide illustrates another mapping.

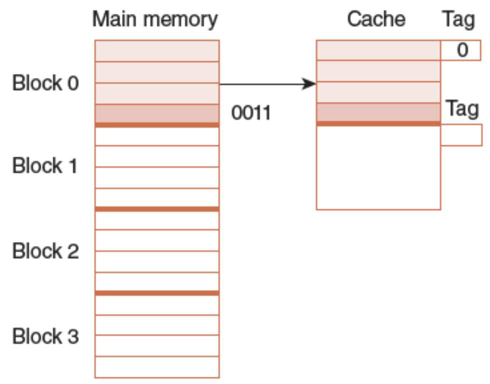
6.4 Cache Memory (9 of 45)



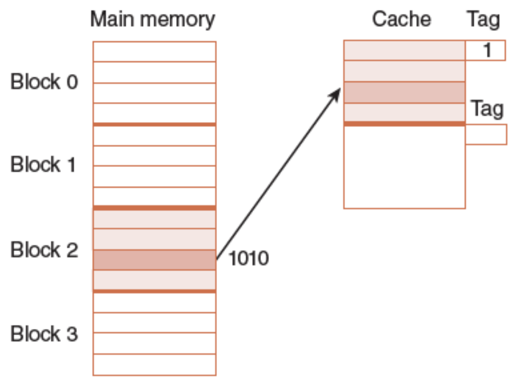
The address 0011 partitioned into fields



The address 1010 partitioned into fields



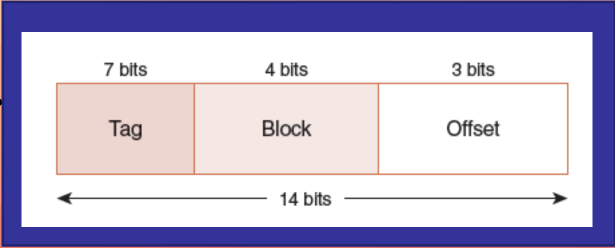
Mapping of block containing
Address 0011 = 0x3



Mapping of Block Containing
Address 1010 = 0xA

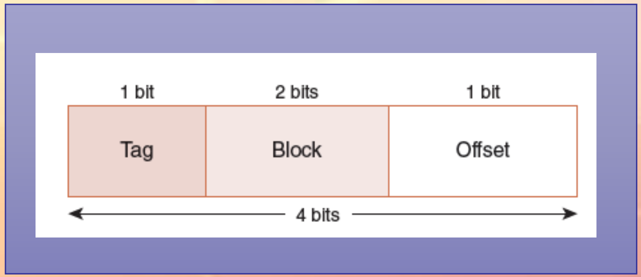
6.4 Cache Memory (10 of 45)

- 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 requires 14 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 (11 of 45)

- Example 6.3: Assume a byte-addressable memory consisting of 16 bytes divided into 8 blocks. Cache contains 4 blocks.

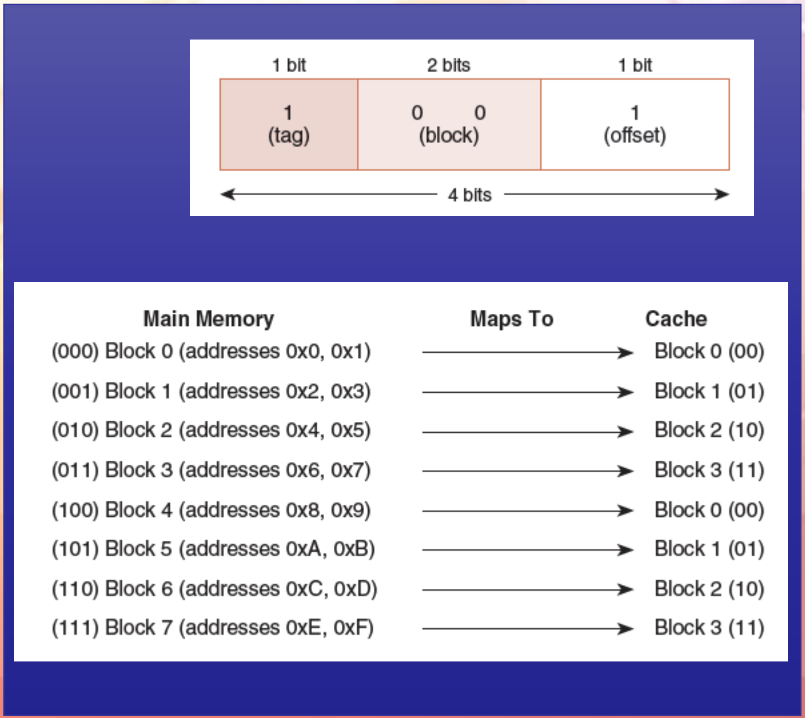


- We know:
  - A memory address has 4 bits.
  - The 4-bit memory address is divided into the fields below.

6.4 Cache Memory (12 of 45)

- Example 6.3: Cont'd.

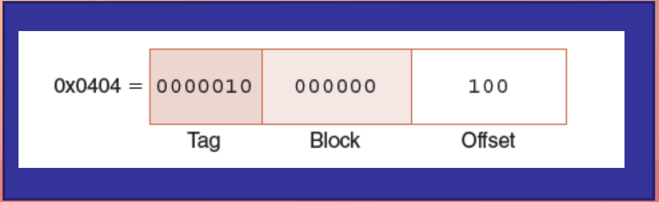
The mapping for memory references is shown below:



Main Memory	Maps To	Cache
(000) Block 0 (addresses 0x0, 0x1)	→	Block 0 (00)
(001) Block 1 (addresses 0x2, 0x3)	→	Block 1 (01)
(010) Block 2 (addresses 0x4, 0x5)	→	Block 2 (10)
(011) Block 3 (addresses 0x6, 0x7)	→	Block 3 (11)
(100) Block 4 (addresses 0x8, 0x9)	→	Block 0 (00)
(101) Block 5 (addresses 0xA, 0xB)	→	Block 1 (01)
(110) Block 6 (addresses 0xC, 0xD)	→	Block 2 (10)
(111) Block 7 (addresses 0xE, 0xF)	→	Block 3 (11)

6.4 Cache Memory (13 of 45)

- 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 (14 of 45)

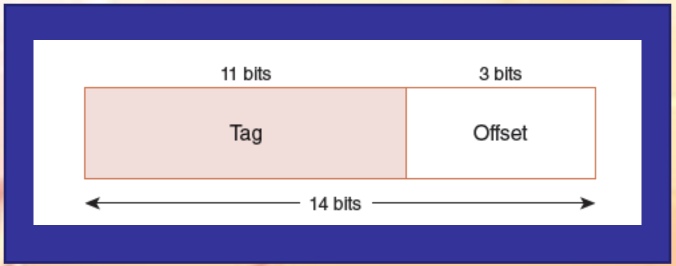
- In summary, direct mapped cache maps main memory blocks in a modular fashion to cache blocks. The mapping depends on:
  - The number of bits in the main memory address (how many addresses exist in main memory).
  - The number of blocks are in cache (which determines the size of the block field).
  - How many addresses (either bytes or words) are in a block (which determines the size of the offset field)?

6.4 Cache Memory (15 of 45)

- Suppose 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 offset.

6.4 Cache Memory (16 of 45)

- 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 (17 of 45)

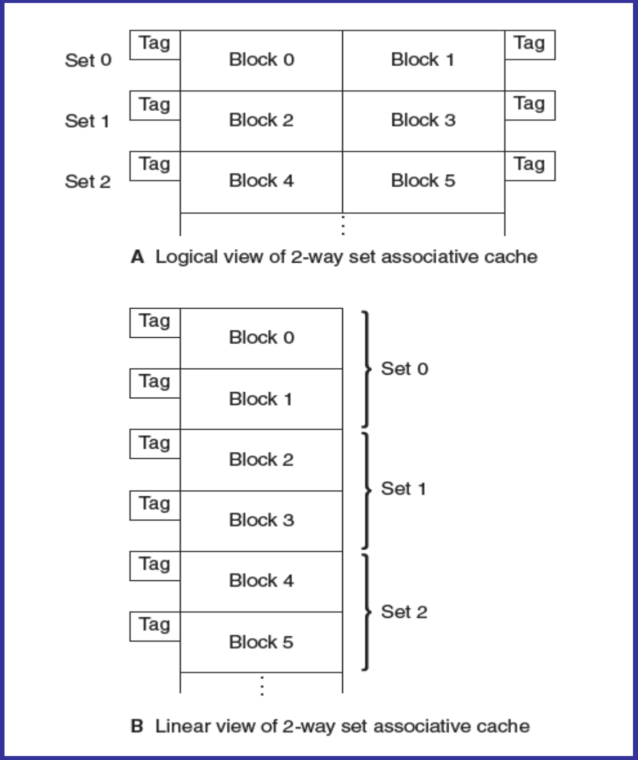
- You will recall that *direct mapped cache* evicts a block whenever another memory reference needs that block.
- With *fully associative cache*, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the *victim block*.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

6.4 Cache Memory (18 of 45)

- ***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, like 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 (19 of 45)

- **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 shown.**
  - **Each set contains two different memory blocks.**

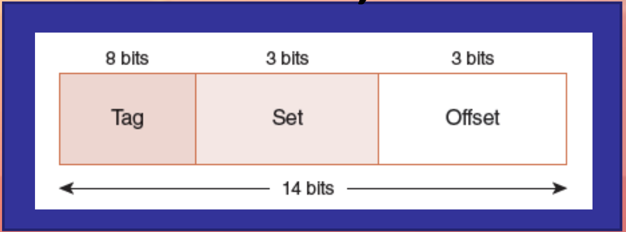

Set 0
Set 1
Set 2
Tag
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
A Logical view of 2-way set associative cache
B Linear view of 2-way set associative cache

6.4 Cache Memory (20 of 45)

- **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 byte 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.**

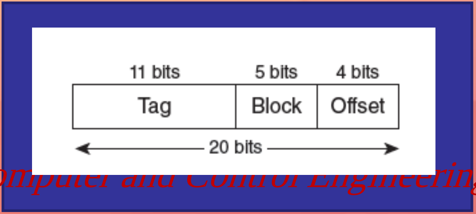
6.4 Cache Memory (21 of 45)

- **Example 6.5: Suppose we are using 2-way set associative mapping with a byte-addressable main memory of 2^{14} bytes and a cache with 16 blocks, where each block contains 8 bytes.**
  - **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.**



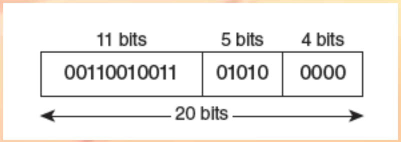
6.4 Cache Memory (22 of 45)

- **Example 6.6: Suppose a byte-addressable memory contains 1MB and cache consists of 32 blocks, where each block contains 16 bytes. Using direct mapping, fully associative mapping, and a 4-way set associative mapping, determine where the main memory address 0x326A0 maps to in cache.**
  - **First note that a main memory address has 20 bits. The main memory address for direct mapped cache is shown below.**



6.4 Cache Memory (23 of 45)

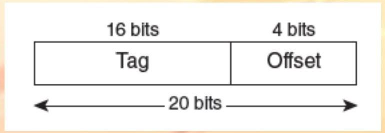
- **Example 6.6:**
  - **If we represent our main memory address 0x326A0 in binary and place the bits into the format, we get:**



  - **So, this address maps to cache block 01010 (or block 10).**

6.4 Cache Memory (24 of 45)

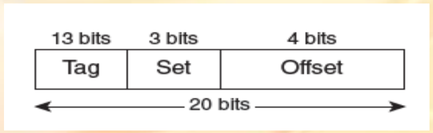
- **Example 6.6: Cont'd.**
  - **If we are using fully associative cache, we have:**



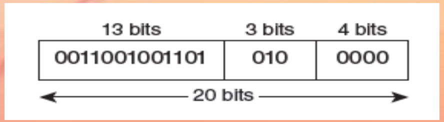
  - **But because it is fully associative, it could map anywhere.**

6.4 Cache Memory (25 of 45)

- **Example 6.6: Cont’d.**
  - **If we are using 4-way set associative cache, we have:**

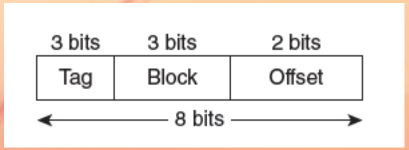


  - **If we divide the main memory address into these fields, we get:**



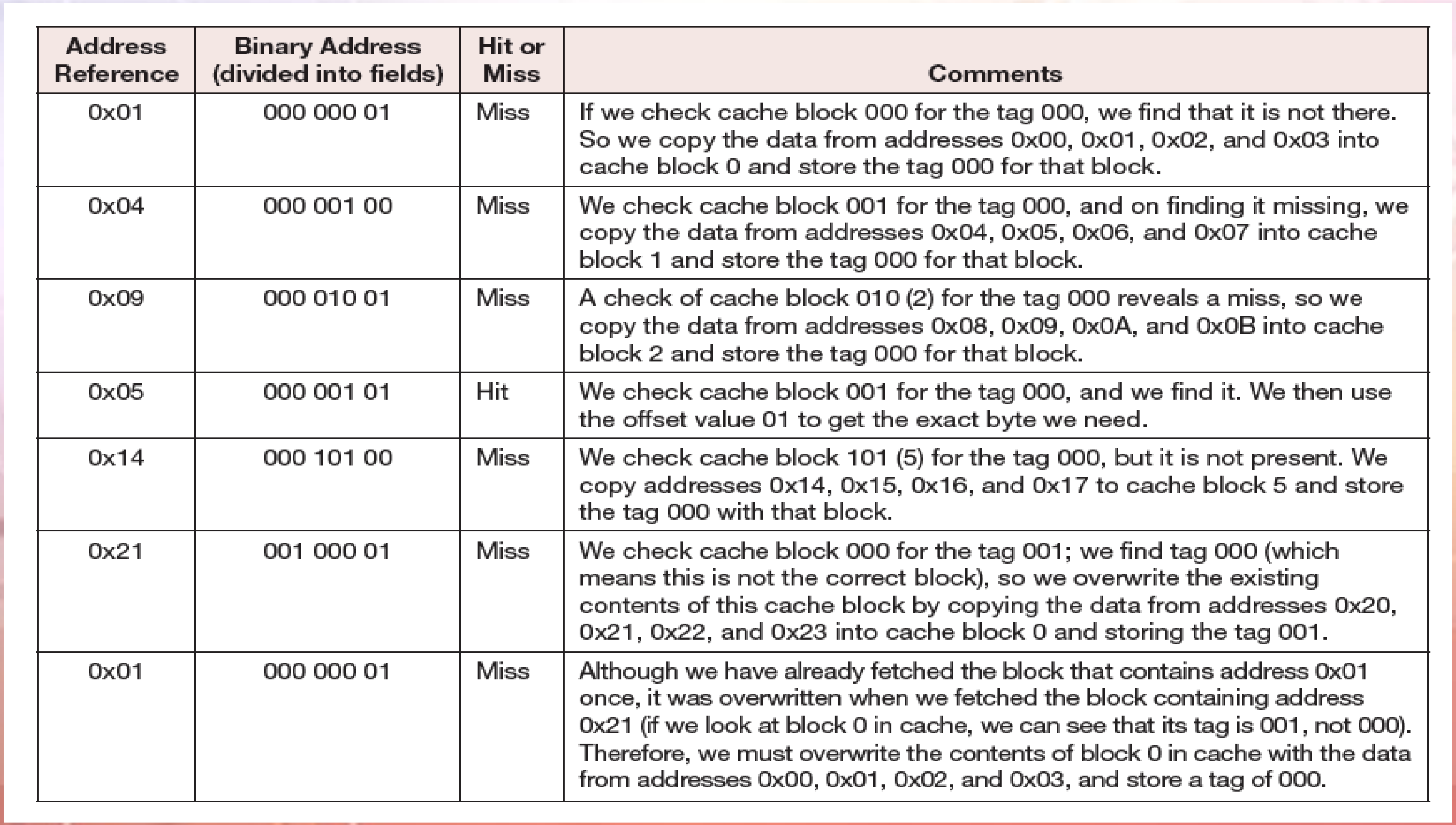
6.4 Cache Memory (26 of 45)

- **Example 6.7: A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach.**
- **The address format for direct mapped cache is:**



Our trace is on the next slide.

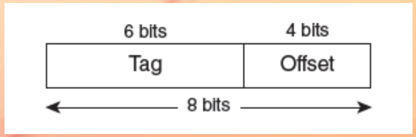
6.4 Cache Memory (27 of 45)



Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000 000 01	Miss	If we check cache block 000 for the tag 000, we find that it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000 for that block.
0x04	000 001 00	Miss	We check cache block 001 for the tag 000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000 for that block.
0x09	000 010 01	Miss	A check of cache block 010 (2) for the tag 000 reveals a miss, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000 for that block.
0x05	000 001 01	Hit	We check cache block 001 for the tag 000, and we find it. We then use the offset value 01 to get the exact byte we need.
0x14	000 101 00	Miss	We check cache block 101 (5) for the tag 000, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 5 and store the tag 000 with that block.
0x21	001 000 01	Miss	We check cache block 000 for the tag 001; we find tag 000 (which means this is not the correct block), so we overwrite the existing contents of this cache block by copying the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 0 and storing the tag 001.
0x01	000 000 01	Miss	Although we have already fetched the block that contains address 0x01 once, it was overwritten when we fetched the block containing address 0x21 (if we look at block 0 in cache, we can see that its tag is 001, not 000). Therefore, we must overwrite the contents of block 0 in cache with the data from addresses 0x00, 0x01, 0x02, and 0x03, and store a tag of 000.

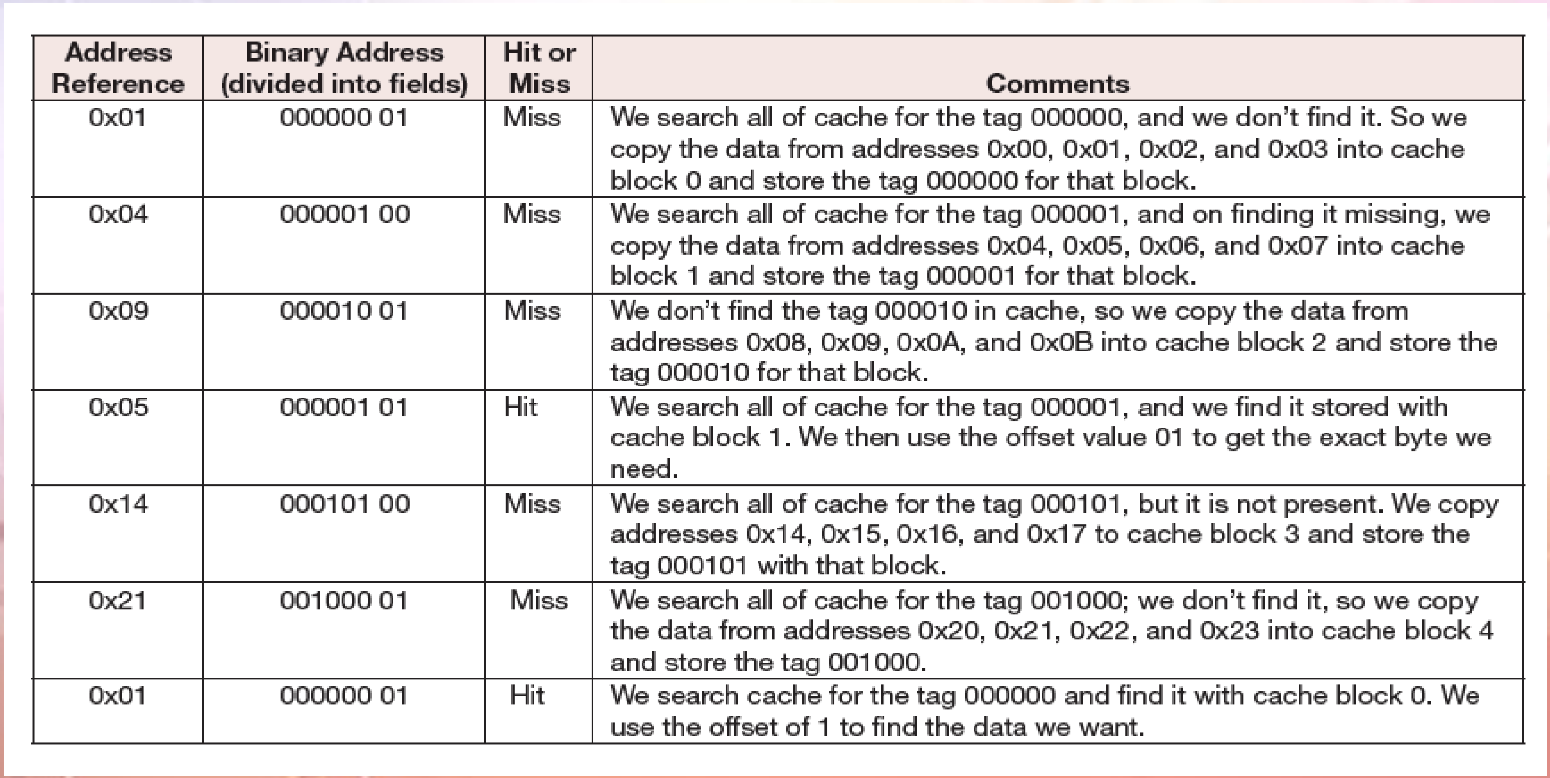
6.4 Cache Memory (28 of 45)

- **Example 6.7: Cont'd. A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach.**
- **The address format for fully associative cache is:**



Our trace is on the next slide.

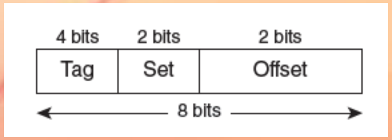
6.4 Cache Memory (29 of 45)



Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000000 01	Miss	We search all of cache for the tag 000000, and we don't find it. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000000 for that block.
0x04	000001 00	Miss	We search all of cache for the tag 000001, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000001 for that block.
0x09	000010 01	Miss	We don't find the tag 000010 in cache, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000010 for that block.
0x05	000001 01	Hit	We search all of cache for the tag 000001, and we find it stored with cache block 1. We then use the offset value 01 to get the exact byte we need.
0x14	000101 00	Miss	We search all of cache for the tag 000101, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 3 and store the tag 000101 with that block.
0x21	001000 01	Miss	We search all of cache for the tag 001000; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 4 and store the tag 001000.
0x01	000000 01	Hit	We search cache for the tag 000000 and find it with cache block 0. We use the offset of 1 to find the data we want.

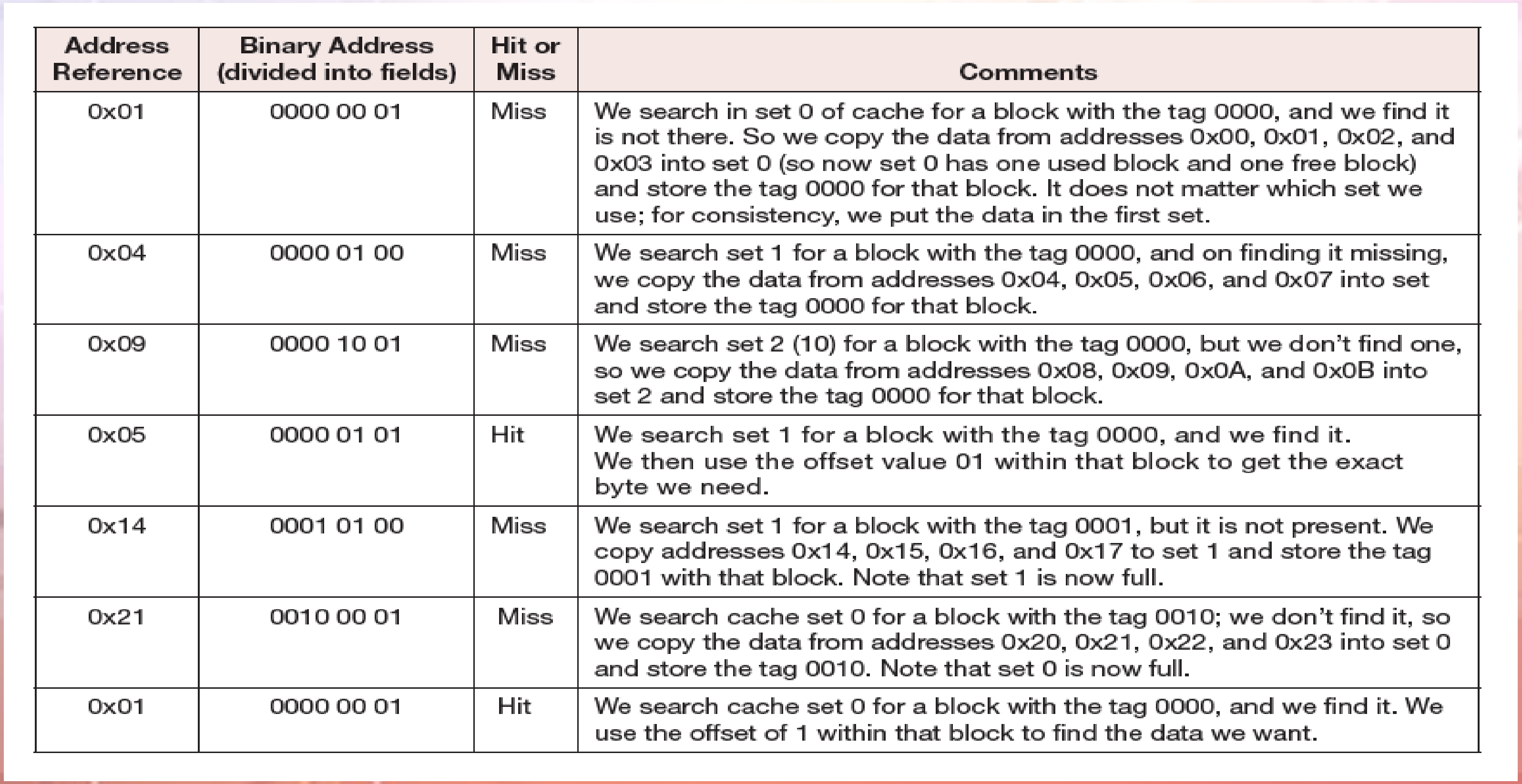
6.4 Cache Memory (30 of 45)

- **EXAMPLE 6.7: Cont'd. A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach.**
- **The address format for 2-way set-associative cache is:**



Our trace is on the next slide.

6.4 Cache Memory (31 of 45)



Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	0000 00 01	Miss	We search in set 0 of cache for a block with the tag 0000, and we find it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into set 0 (so now set 0 has one used block and one free block) and store the tag 0000 for that block. It does not matter which set we use; for consistency, we put the data in the first set.
0x04	0000 01 00	Miss	We search set 1 for a block with the tag 0000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into set and store the tag 0000 for that block.
0x09	0000 10 01	Miss	We search set 2 (10) for a block with the tag 0000, but we don't find one, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into set 2 and store the tag 0000 for that block.
0x05	0000 01 01	Hit	We search set 1 for a block with the tag 0000, and we find it. We then use the offset value 01 within that block to get the exact byte we need.
0x14	0001 01 00	Miss	We search set 1 for a block with the tag 0001, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to set 1 and store the tag 0001 with that block. Note that set 1 is now full.
0x21	0010 00 01	Miss	We search cache set 0 for a block with the tag 0010; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into set 0 and store the tag 0010. Note that set 0 is now full.
0x01	0000 00 01	Hit	We search cache set 0 for a block with the tag 0000, and we find it. We use the offset of 1 within that block to find the data we want.

Week 3 : 1/3/2023 2013

Week 3 : 1/3/2023 2019

6.4 Cache Memory (32 of 45)

- **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.**

- **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 produce.**

6.4 Cache Memory (33 of 45)

- 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.

- The disadvantage of this approach is its complexity: LRU must maintain an access history for each block, which ultimately slows down the cache.

6.4 Cache Memory (34 of 45)

- *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, but it never thrashes.

6.4 Cache Memory (35 of 45)

- **The performance of hierarchical memory is measured by its *Effective Access Time (EAT)*.**

- **EAT is a weighted average that considers the hit ratio and relative access times of successive levels of memory.**

- **The EAT for a two-level memory is given by:**

$$EAT = H \times Access_C + (1 - H) \times Access_{MM}$$

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 (36 of 45)

- **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%.**

- **Suppose access to cache and main memory occurs concurrently (the accesses overlap).**

- **The EAT is** $EAT = H \times Access_C + (1 - H) \times Access_{MM}$

$$= 0.99(10ns) + 0.01(200ns)$$

$$= 9.9ns + 2ns$$

$$= 11ns$$

6.4 Cache Memory (37 of 45)

- **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%.**
- **If the accesses do not overlap,**

the EAT = 0.99(10ns) + 0.01(10ns + 200ns)
= 9.9ns + 2.01ns = 12ns

- **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 (38 of 45)

- **Caching is depending upon programs exhibiting good locality.**
  - **Some object-oriented programs have poor locality owing to their complex, dynamic structures.**
  - **Arrays stored in column-major rather than row-major order can be problematic for certain cache organizations.**
- **With poor locality, caching can cause performance degradation rather than performance improvement.**

6.4 Cache Memory (39 of 45)

- **Cache replacement policies must consider *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.**
- **Write back (also called *copyback*) updates memory only when the block is selected for replacement.**

6.4 Cache Memory (40 of 45)

- The disadvantage of write through is that memory must be updated with each cache write, which slows down the access time on updates. This slowdown is usually negligible, because most accesses tend to be reads, not writes.

- 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.

6.4 Cache Memory (41 of 45)

- The cache we have been discussing is called a *unified* or *integrated* cache where both instructions and data are cached.

- Many modern systems employ separate caches for data and instructions.
  - This is called a *Harvard* cache.

- The separation of data from instructions provides better locality, at the cost of greater complexity.
  - Simply making the cache larger provides about the same performance improvement without the complexity.

6.4 Cache Memory (42 of 45)

- **Cache performance can also be improved by adding a small associative cache to hold blocks that have been evicted recently.**
  - **This is called a *victim cache*.**
- **A *trace cache* is a variant of an instruction cache that holds decoded instructions for program branches, giving the illusion that noncontiguous instructions are contiguous.**

6.4 Cache Memory (43 of 45)

- **Most of today's small systems employ multilevel cache hierarchies.**
- **The levels of cache form their own small memory hierarchy.**
- **Level 1 cache (8KB to 64KB) is situated on the processor itself.**
  - **Access time is typically about 4ns.**
- **Level 2 cache (64KB to 2MB) may be on the motherboard, or on an expansion card.**
  - **Access time is usually around 15–20ns.**

6.4 Cache Memory (44 of 45)

- **In systems that employ three levels of cache, the Level 2 cache is placed on the same die as the CPU (reducing access time to about 10ns).**

- **Accordingly, the Level 3 cache (2MB to 256MB) refers to cache that is situated between the processor and main memory.**

- **Once the number of cache levels is determined, the next thing to consider is whether data (or instructions) can exist in more than one cache level.**

6.4 Cache Memory (45 of 45)

- If the cache system used an *inclusive* cache, the same data may be present at multiple levels of cache.
- *Strictly inclusive* caches guarantee that all data in a smaller cache also exists at the next higher level.
- *Exclusive* caches permit only one copy of the data.
- The tradeoffs in choosing one over the other involve weighing the variables of access time, memory size, and circuit complexity.

Week 4 : 8/3/2023 2019

6.5 Virtual Memory (1 of 26)

- 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.
- Using virtual memory, your computer addresses more main memory than it has, and it uses the hard drive to hold the excess. This area on the hard drive is called a page file, because it holds chunks of main memory on the hard drive. The easiest way to think about virtual memory is to conceptualize it as an imaginary memory location in which all addressing issues are handled by the operating system.
- If a system uses paging, virtual memory partitions main memory into individually managed page frames, that are written (or paged) to disk when they are not immediately needed.

6.5 Virtual Memory (2 of 26)

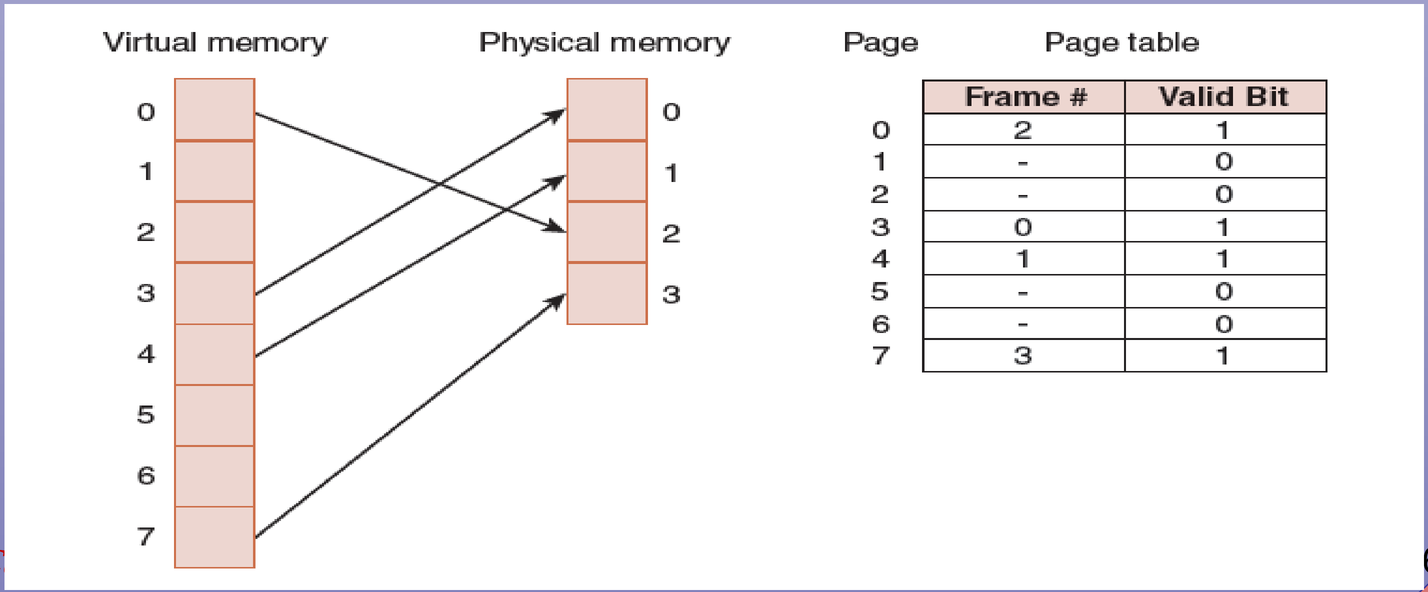
- ***Virtual address***—The logical or program address that the process uses. Whenever the CPU generates an address, it is always in terms of virtual address space.
- ***Physical address*** is the actual memory address of physical memory.
- ***Mapping***—The mechanism by which virtual addresses are translated into physical ones (very similar to cache mapping).
- ***Page frames***—The equal-size chunks or blocks into which main memory (physical memory) is divided.
- ***Pages***—The chunks or blocks into which virtual memory (the logical address space) is divided, each equal in size to a page frame. Virtual pages are stored on disk until needed.
- ***Paging***—The process of copying a virtual page from disk to a page frame in main memory.
- **Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager.**
- ***Page faults*** occur when a logical address requires that a page be brought in from disk (a requested page is not in main memory).
- ***Memory fragmentation*** occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

6.5 Virtual Memory (3 of 26)

- 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.
- Virtual memory can be implemented with different techniques, including paging, segmentation, or a combination of both, but paging is the most popular.
- The success of paging, like that of cache, is dependent on the locality principle

6.5 Virtual Memory (4 of 26)

- 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.
- The page table stores the physical location of each virtual page of the process.
- The page table has N rows, where N is the number of virtual pages in the process.



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 (5 of 26)

- The page table indicates pages of the process which are not in main memory by setting a valid bit to 0, otherwise it is set to 1.
- Additional fields may be added:
  - like a dirty bit (or a modify bit) : to indicate whether the page has been changed. This makes returning, if it is not modified, it does not need to be rewritten to disk.
  - Another bit (the usage bit) to indicate the page usage. This bit is set to 1 whenever the page is accessed. After a certain time period, the usage bit is set to 0. If the page is referenced again, the usage bit is set to 1. However, if the bit remains 0, this indicates that the page has not been used for a period, and the system might benefit by sending this page out to disk. By doing so, the system frees up this page’s location for another page that the process eventually needs

Week 4 : 8/3/2023 2013

6.5 Virtual Memory (5 of 26) – continued

- **When a process generates a virtual address, the operating system translates it into a physical memory address. *(let's assume that we have no cache memory for the moment.)***
  - **For example, from a program viewpoint, we see the final byte of a 10-byte program as address 0x09, assuming 1-byte instructions and 1-byte addresses and a starting address of 0x00. However, when loaded into memory, the logical address 0x09 (perhaps a reference to the label X in an assembly language program) may reside in physical memory location 0x39, implying that the program was loaded starting at physical address 0x30. There must be an easy way to convert the logical, or virtual, address 0x09 to the physical address 0x39.**
- **To accomplish this, the virtual address is divided into two fields: A *page field*, and an *offset field*, to represent the offset within that page where the requested data is located.**
- **Like cache blocks, page sizes are usually powers of 2; this simplifies the extraction of page numbers and offsets from virtual addresses.**

6.5 Virtual Memory (5 of 26) – continued

- **To access data at a given virtual address, the system performs the following 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.**

6.5 Virtual Memory (5 of 26) – continued

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 = 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 B2.

6.5 Virtual Memory (5,6 of 26) – continued

2. **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 the offset in the physical page frame by adding the offset to the frame number for the given virtual page.**

To access data at a given virtual address, the system performs the following steps: Extract the

1. 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).

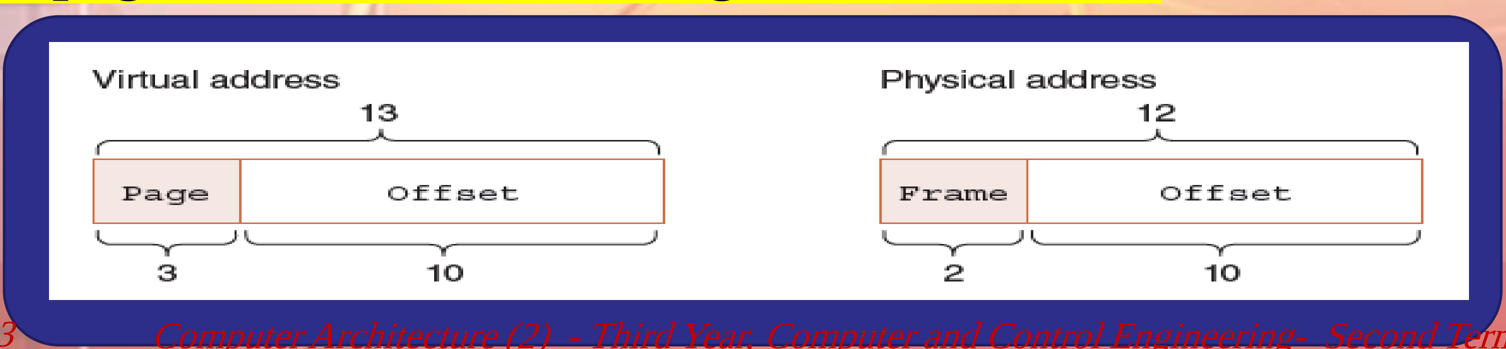
B. Check the valid bit for that page.

1. **If the valid bit = 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 B2.**

2. **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 the offset in the physical page frame by adding the offset to the frame number for the given virtual page.**

6.5 Virtual Memory (7 of 26)

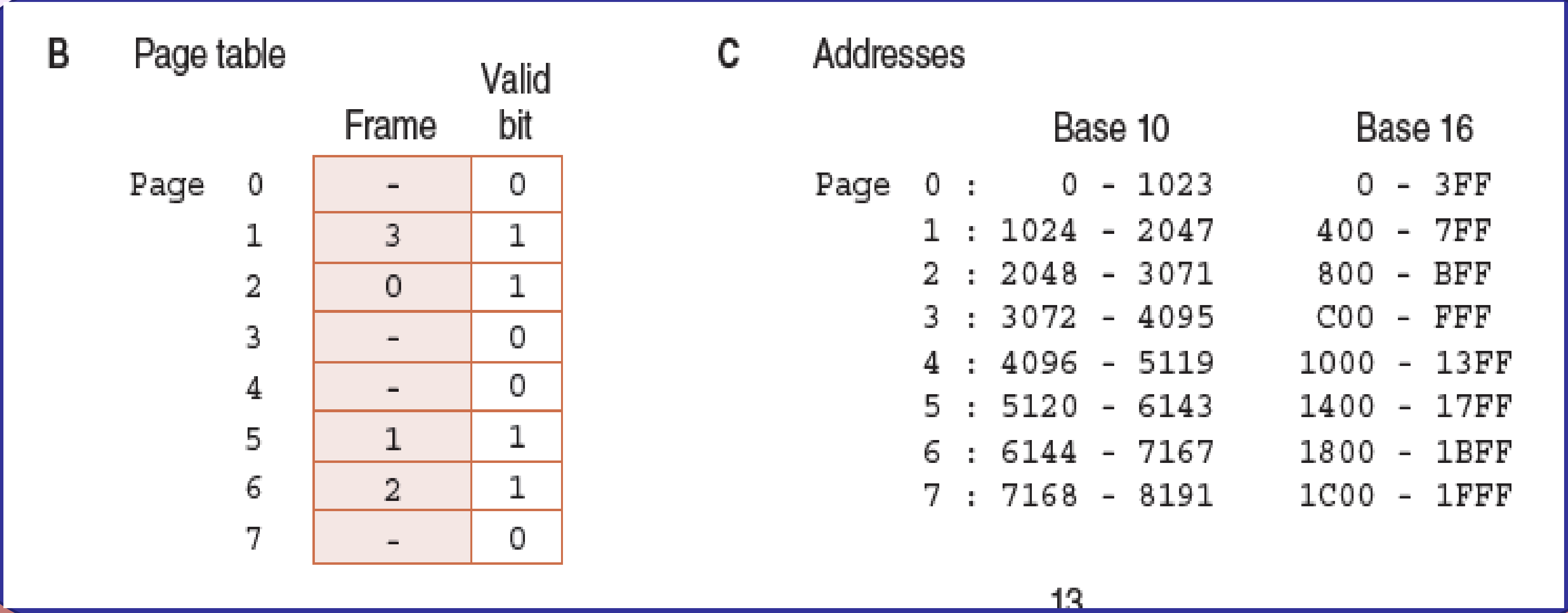
- As an example, suppose a system has a virtual address space of 8K and a physical address space of 4K, and the system uses byte addressing. Page size 1K.
  - We have $2^{13}/2^{10} = 2^3$ virtual pages.
- A virtual address has 13 bits (8K = 2^{13}) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first 2 bits for the page frame and the trailing 10 bits the offset.



6.5 Virtual Memory (8 of 26)

- Suppose we have the page table shown below.
- What happens when the CPU generates address:

$5459_{10} = 1010101010011_2 = 0x1553$?



B 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

C Addresses

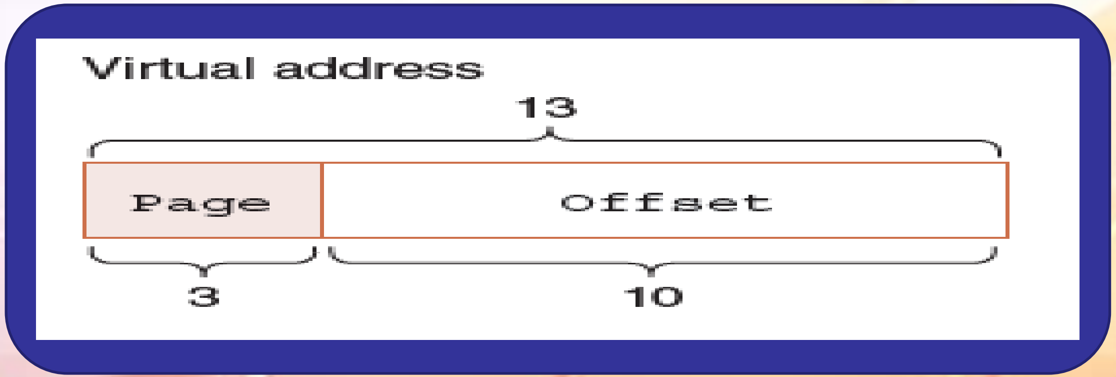
Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

13

6.5 Virtual Memory (9 of 26)

- What happens when the CPU generates address

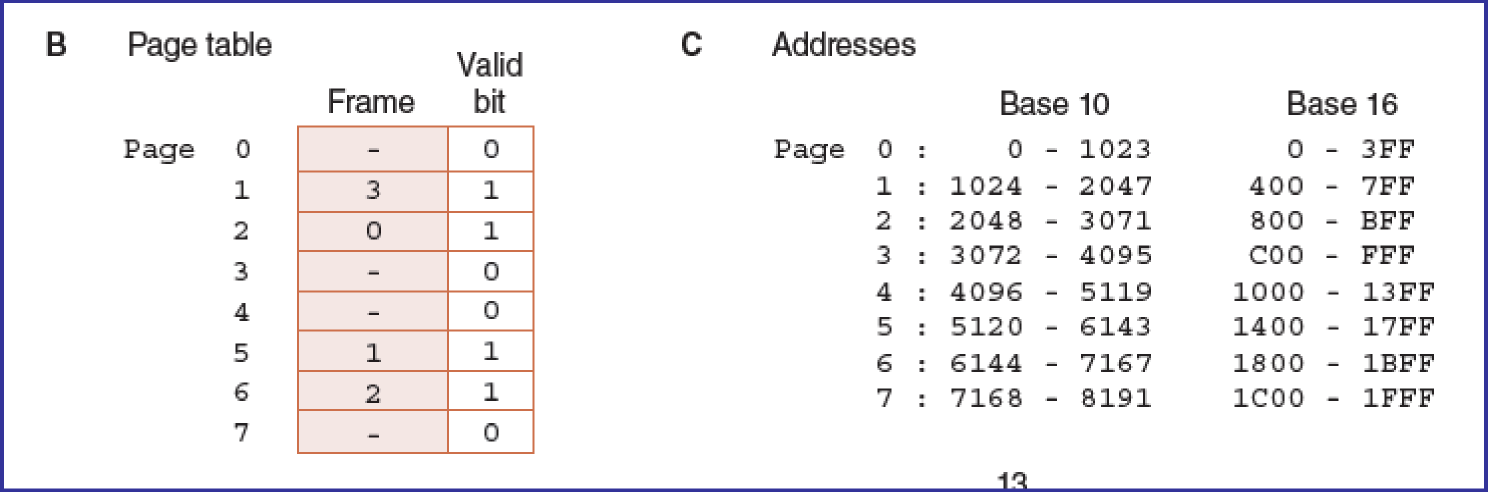
  $5459_{10} = 1010101010011_2 = 0x1553$?



- The high-order 3 bits of the virtual address, 101 (5_{10}), provide the page number in the page table.

6.5 Virtual Memory (10 of 26)

- **The address 1010101010011_2 is converted to physical address 010101010011_2 = 0x1363 because the page field 101 is replaced by frame number 01 through a lookup in the page table.**



B 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

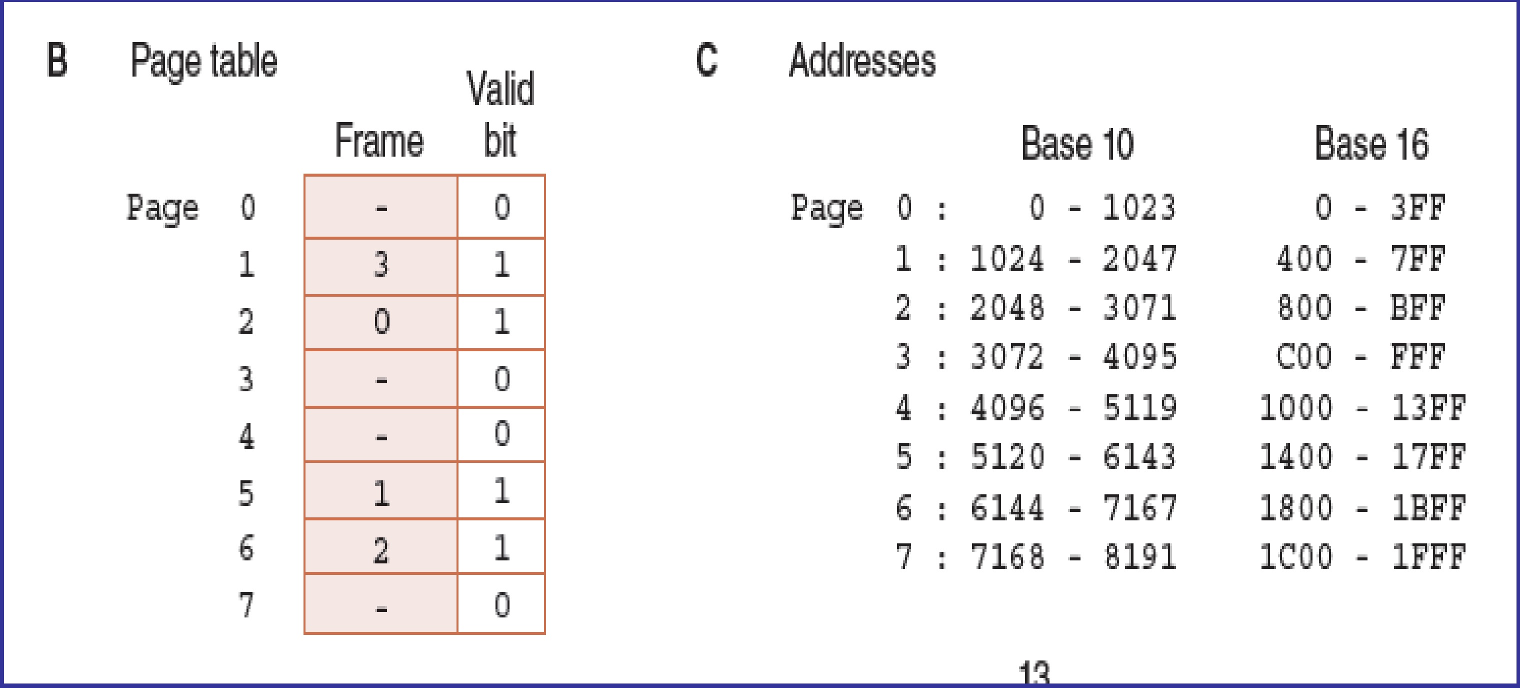
C Addresses

		Base 10	Base 16
Page	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

13

6.5 Virtual Memory (11 of 26)

- What happens when the CPU generates address 1000000000100_2?



B 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

C 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

13

6.5 Virtual Memory (12 of 26)

- **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 must consider page table access time.**
- **For each memory access that the processor generates, there must now be two physical memory accesses—one to reference the page table and one to reference the actual data we wish to access**
- **Suppose a main memory access takes 200ns, the page fault rate is 1%, (that is, 99% of the time we find the page we need in memory). Assume it takes 10ms to load a page from disk. (This time of 10ms includes the time necessary to transfer the page into memory, update the page table, and access the data).**

EAT = 0.99 (200ns + 200ns) + 0.01(10ms) = 100,396 ns

6.5 Virtual Memory (13 of 26)

- **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.**

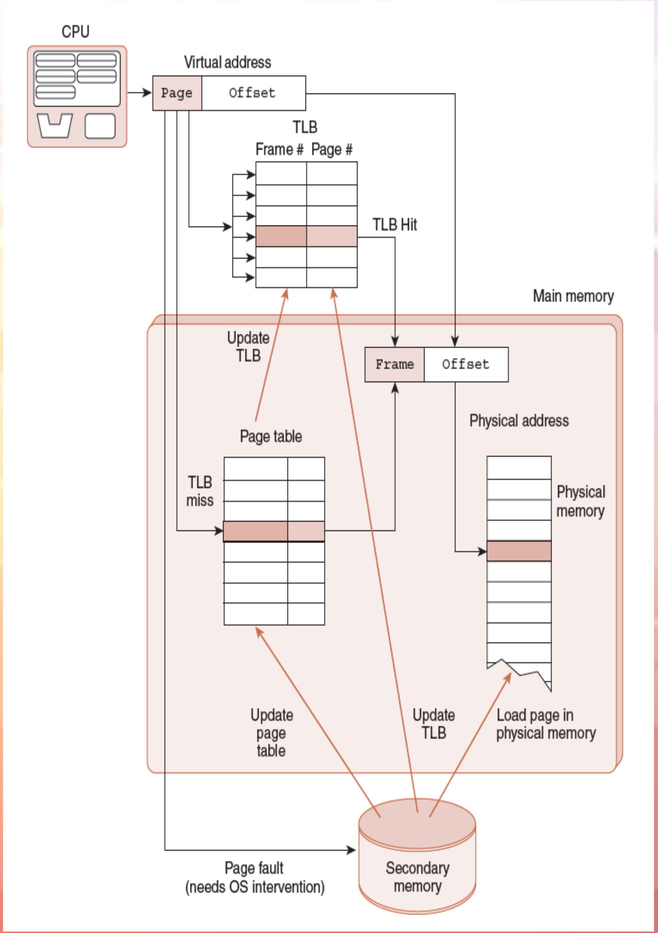
- **Because page tables are read constantly, it makes sense to keep them in a special cache called a *Translation Look-aside Buffer* (TLB).**

- **TLBs are a special associative cache that stores the mapping of virtual pages to physical pages.**

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

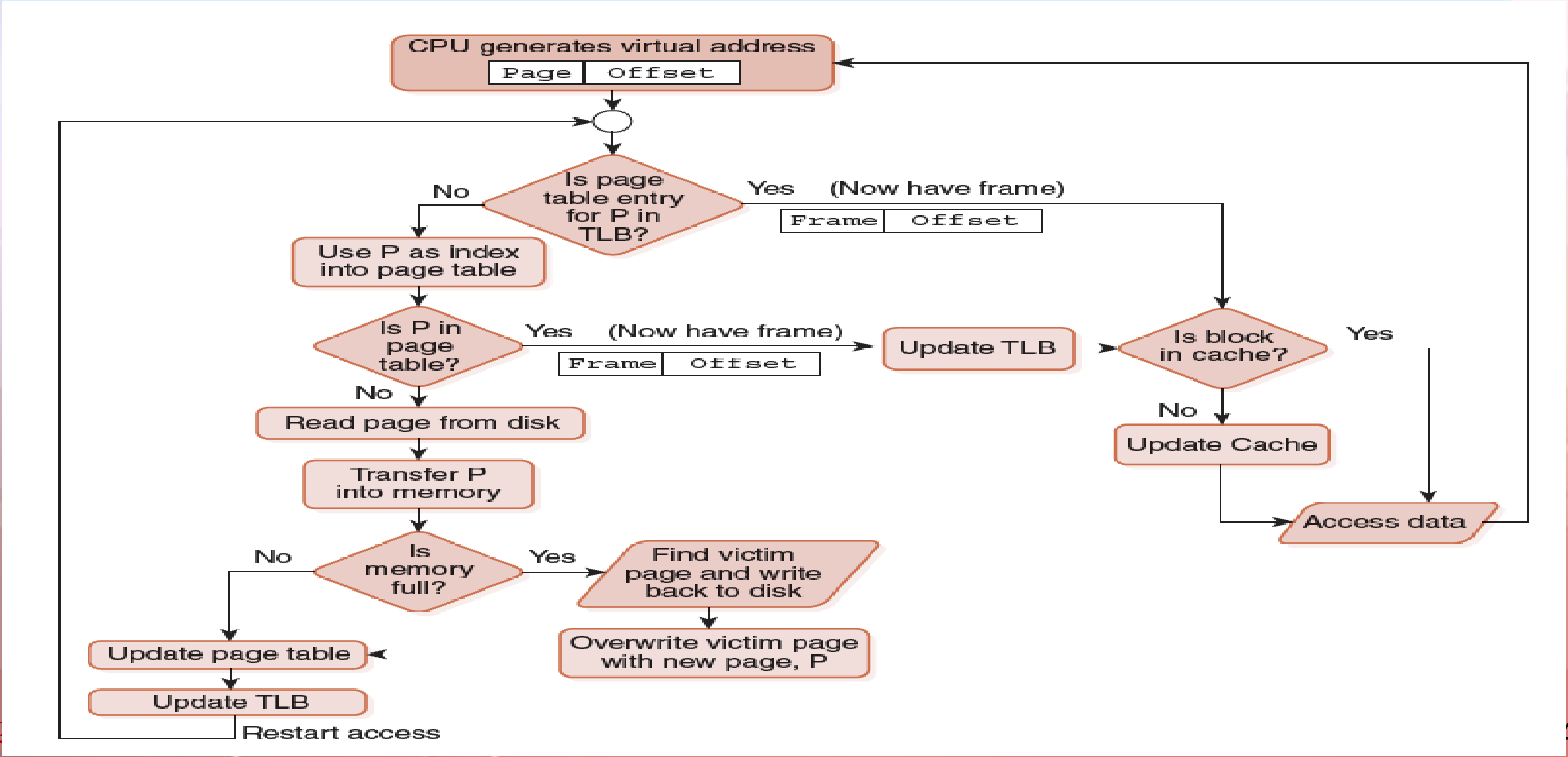
6.5 Virtual Memory (14 of 26)

- TLB lookup process
  - Extract the page number from the virtual address.
  - Extract the offset from the virtual address.
  - Search for the virtual page number in the TLB.
  - 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.
  - 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.
  - 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 (15 of 26)

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



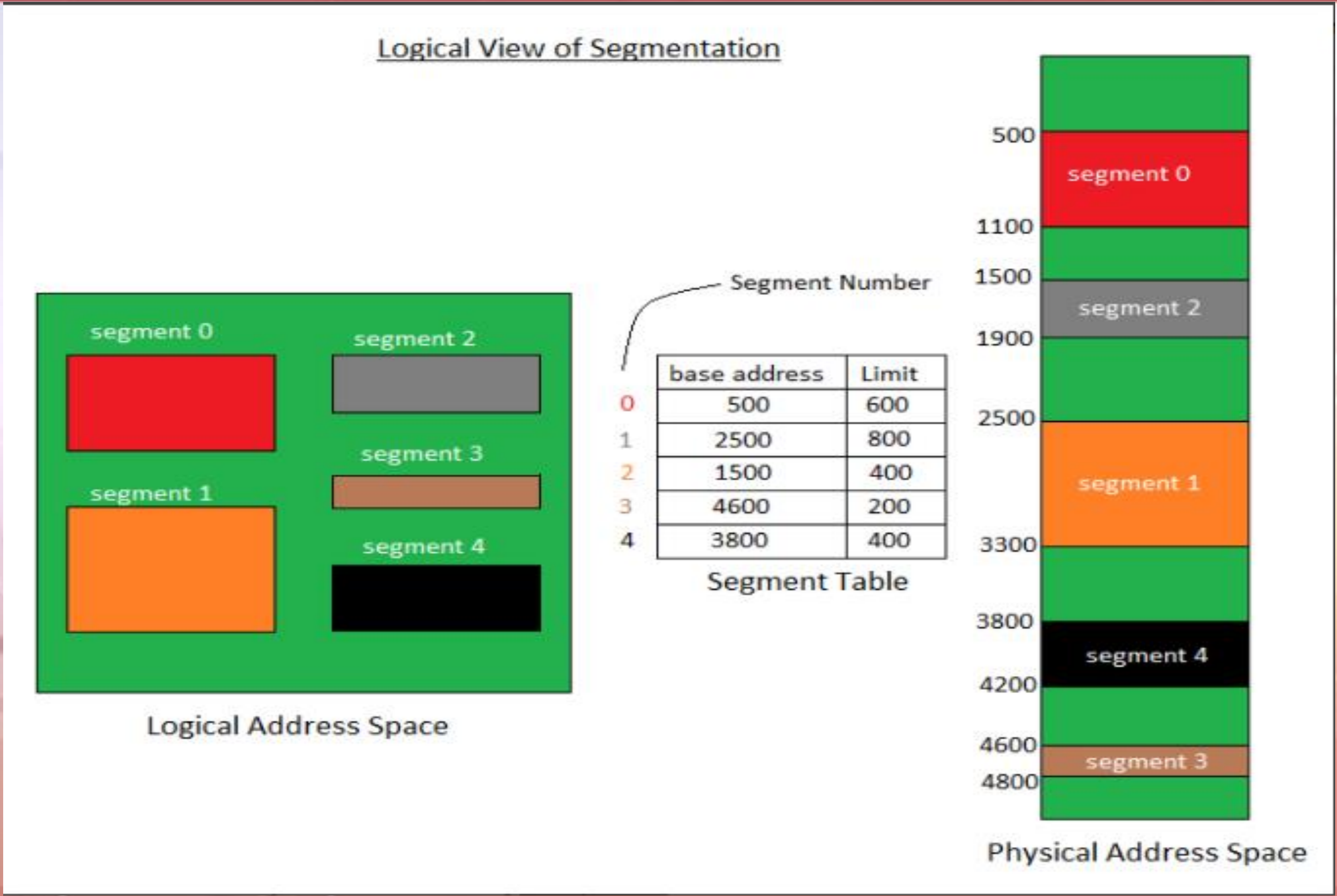
6.5 Virtual Memory (16 of 26)

- **Another approach to virtual memory is the use of *segmentation*.**
- **Instead of dividing the virtual address space into equal, fixed-size pages, and the physical address space into equal-size page frames, the virtual address space is divided into logical, variable-length units, or segments. often under the control of the programmer**
- **Physical memory isn't really divided or partitioned into anything. When a segment needs to be copied into physical memory, the operating system looks for a chunk of free memory large enough to store the entire segment. Each segment has a base address, indicating where it is in memory, and a bounds limit, indicating its size. Each program, consisting of multiple segments, now has an associated segment table instead of a page table.**
- **This segment table is simply a collection of the base/bound's pairs for each segment.**

6.5 Virtual Memory (16 of 26) - Continued

- Memory accesses are translated by providing a segment number and an offset within the segment.
- Error checking is performed to make sure the offset is within the allowable bound. If it is, then the base value for that segment (found in the segment table) is added to the offset, yielding the actual physical address.
- Because paging is based on a fixed-size block and segmentation is based on a logical block, protection and sharing are easier using segmentation.
- For example, the virtual address space might be divided into a code segment, a data segment, a stack segment, and a symbol table segment, each of a different size. It is much easier to say, "I want to share all my data, so make my data segment accessible to everyone" than it is to say, "OK, in which pages does my data reside, and now that I have found those four pages, let's make three of the pages accessible, but only half of that fourth page accessible."

Computer Architecture (2)
بنية الحاسب (٢)



base address	Limit
500	600
2500	800
1500	400
4600	200
3800	400

6.5 Virtual Memory (17 of 26)

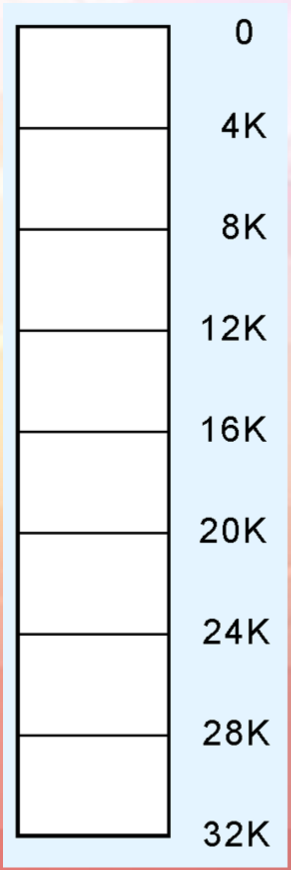
- **Both paging and segmentation can cause fragmentation.**

- **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 (18 of 26)

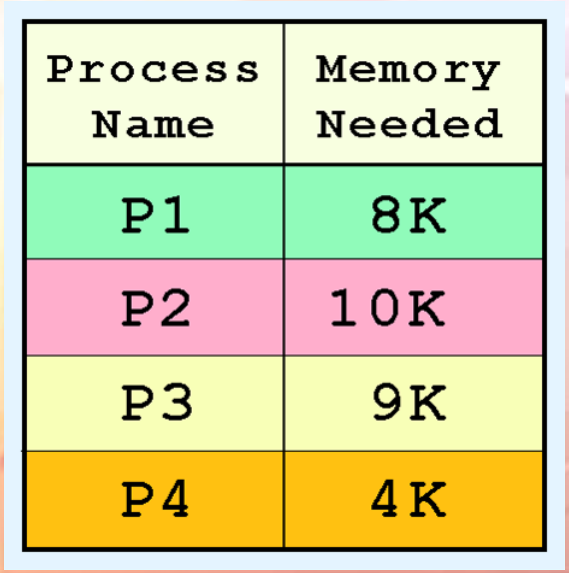
- Consider a small computer having 32K of memory.
- The 32K memory is divided into 8-page frames of 4K each.
- A schematic of this configuration is shown at the right.
- The numbers at the right are memory frame addresses.



6.5 Virtual Memory (19 of 26)

- Suppose there are four processes waiting to be loaded into the system with memory requirements as shown in the table.

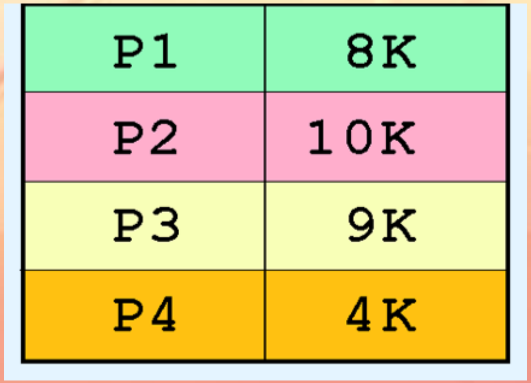
- We observe that these processes require 31K of memory.



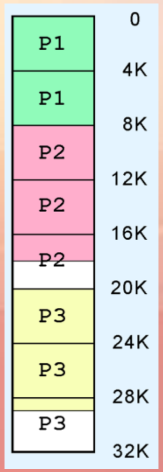
Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

6.5 Virtual Memory (20 of 26)

- **When the first three processes are loaded, memory looks as shown in the figure**
- **All the frames are occupied by three of the processes.**



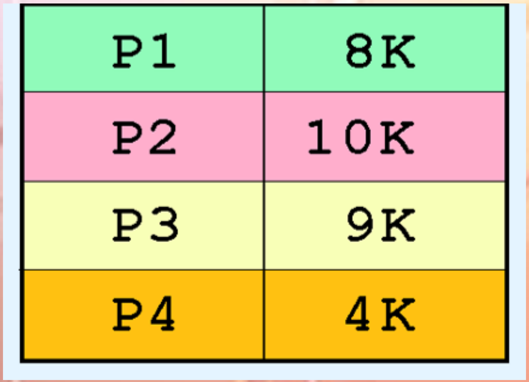
P1	8K
P2	10K
P3	9K
P4	4K



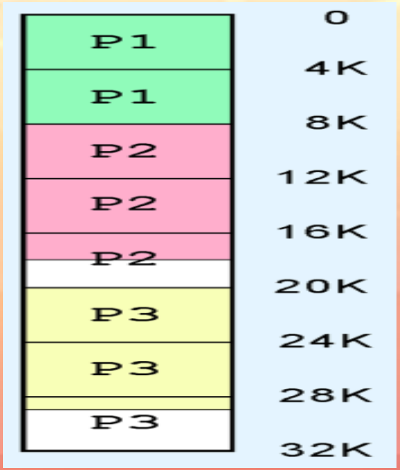
6.5 Virtual Memory (21 of 26)

- Even though there are enough free bytes in memory to load the fourth process, P4 must wait for one of the other three to terminate, because there are no unallocated frames.

- This is an example of *internal fragmentation*.

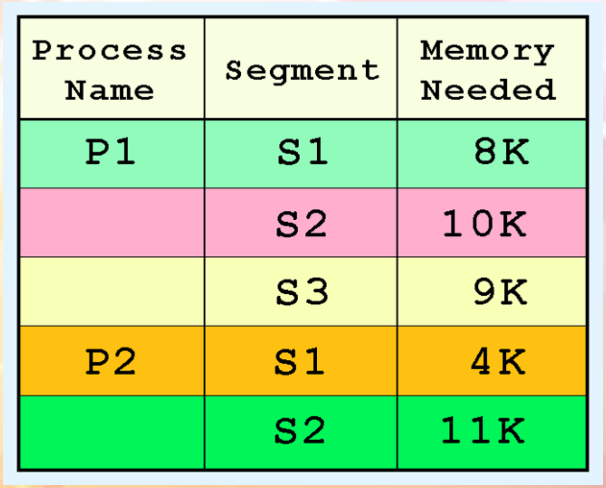


P1	8K
P2	10K
P3	9K
P4	4K



6.5 Virtual Memory (22 of 26)

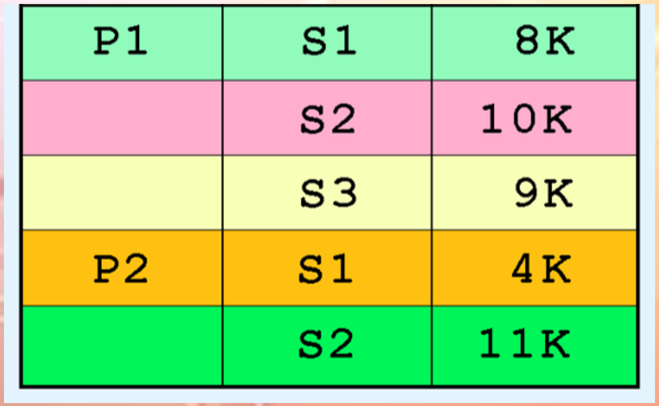
- ❑ Suppose that instead of frames, our 32K system uses segmentation.
- ❑ The memory segments of two processes is shown in the table at the right.
- ❑ The segments can be allocated anywhere in memory.



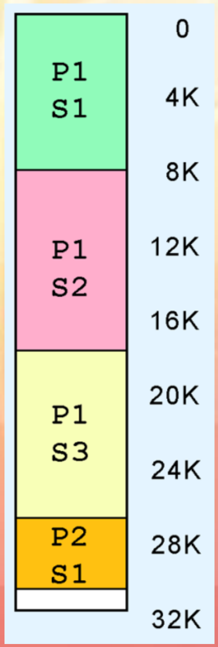
Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

6.5 Virtual Memory (23 of 26)

- **All the segments of P1 and one of the segments of P2 are loaded as shown at the right.**
- **Segment S2 of process P2 requires 11K of memory, and there is only 1K free, so it waits.**

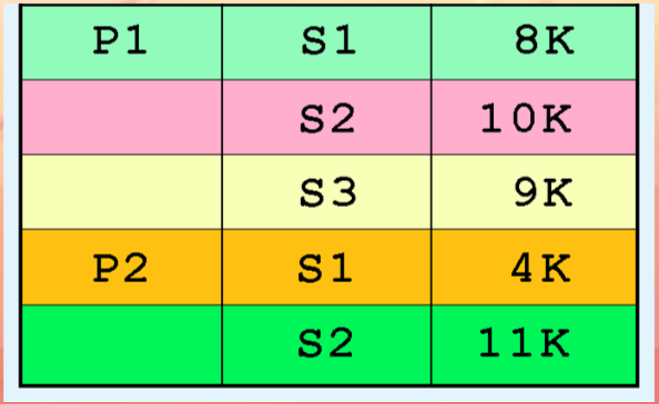


P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

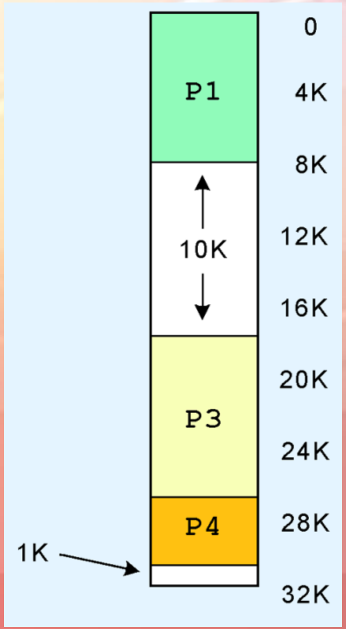


6.5 Virtual Memory (24 of 26)

- Eventually, Segment 2 of Process 1 is no longer needed, so it is unloaded giving 11K of free memory.
- But Segment 2 of Process 2 cannot be loaded because the free memory is not contiguous.



P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

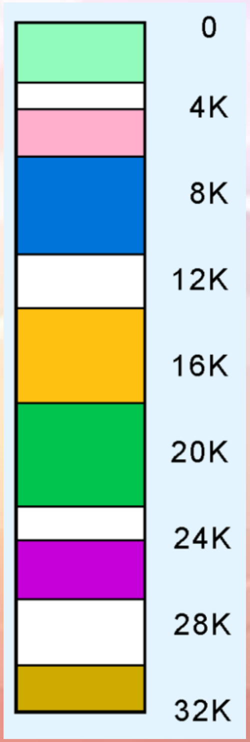


6.5 Virtual Memory (25 of 26)

- Over time, the problem gets worse, resulting in small unusable blocks scattered throughout physical memory.

- This is an example of *external fragmentation*.

- Eventually, this memory is recovered through compaction, and the process starts over.



6.5 Virtual Memory (26 of 26)

- 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.

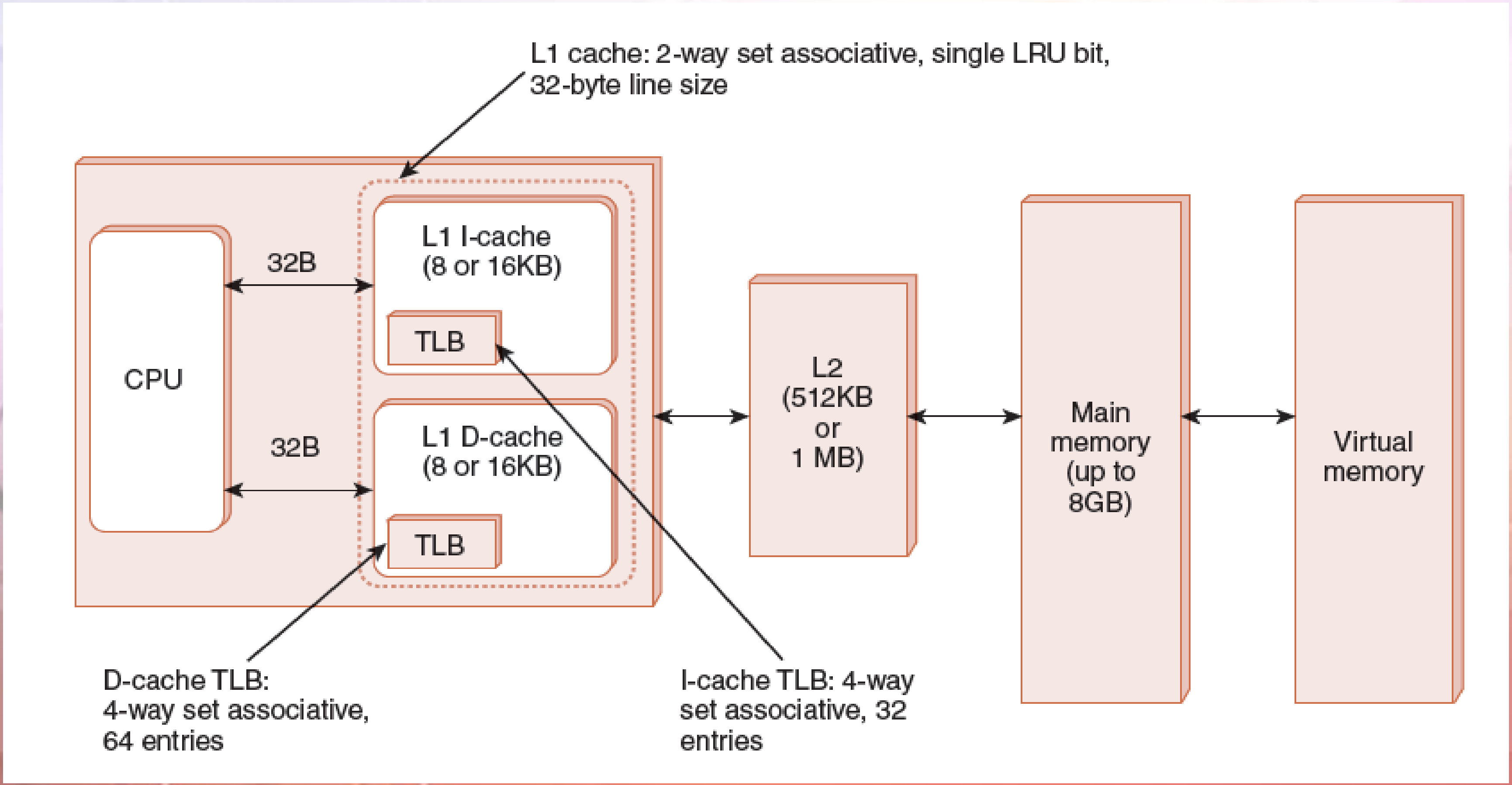
- Each segment has a page table. This means that a memory address will have three fields, one for the segment, another for the page, and a third for the offset.

6.6 A Real-World Example (1 of 2)

- 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 (2 of 2)


L1 cache: 2-way set associative, single LRU bit, 32-byte line size
CPU
32B
32B
L1 I-cache (8 or 16KB)
TLB
L1 D-cache (8 or 16KB)
TLB
L2 (512KB or 1 MB)
Main memory (up to 8GB)
Virtual memory
D-cache TLB: 4-way set associative, 64 entries
I-cache TLB: 4-way set associative, 32 entries

Conclusion (1 of 2)

- ✓ 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.

Conclusion (2 of 2)

- ✓ **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 consider what to do with dirty blocks.**
- ✓ **All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.**

Computer Architecture (2)
بنية الحاسب (٢)

Thank You!