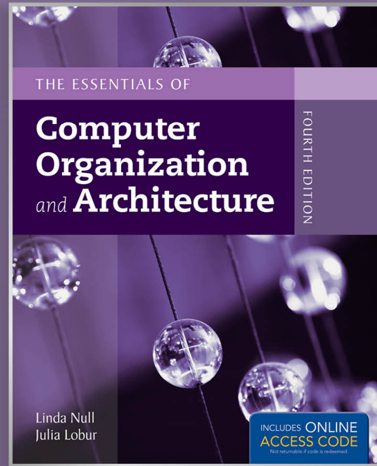
Chapter 6
Memory



© Odua Images/ShutterStock, Inc. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com

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, paging, memory segmentation, 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

- There are two kinds of main memory: *random access memory, RAM* (read-write memory)*, 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

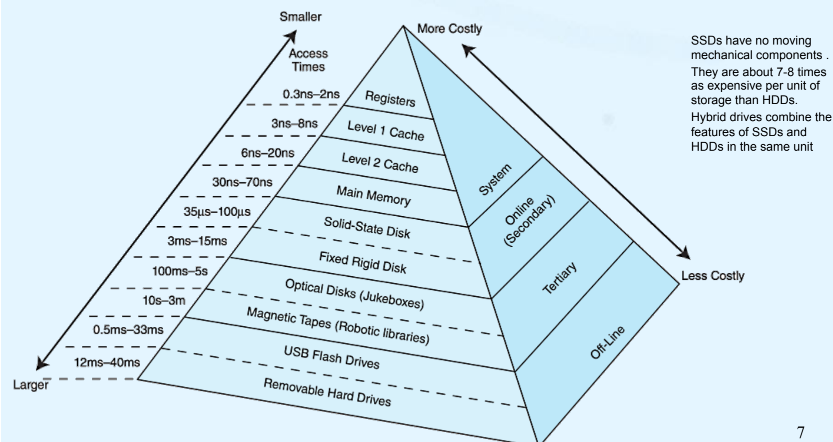
- SRAM consists of circuits similar to the D flip-flop that we studied in Chapter 3.
- SRAM is very fast memory and it doesn't need to be refreshed like DRAM does. It is used to build cache memory, which we will discuss in detail later.
- ROM also does not need to be refreshed, either. In fact, it needs 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.

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:



SSDs have no moving mechanical components . They are about 7-8 times as expensive per unit of storage than HDDs. Hybrid drives combine the features of SSDs and HDDs in the same unit

6.3 The Memory Hierarchy

- 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

- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache (a special high-speed (high-cost) memory).
- 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. A request for the content at location *X* fetches data "around" *X* (..., *X*-2, *X*-1, *X*, *X*+1, *X*+2, ...) 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 block of data is copied after a miss 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.

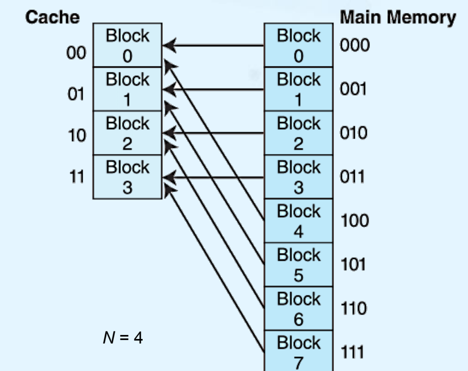
6.4 Cache Memory

- 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.
- 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.

The next slide illustrates this mapping.

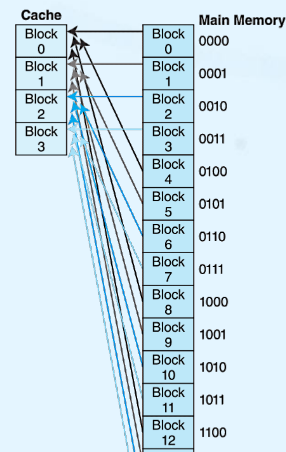
6.4 Cache Memory

- Direct mapping of main memory blocks to cache blocks
  - With direct mapped cache consisting of N blocks of cache, block X of main memory maps to cache block $Y = X \bmod N$.



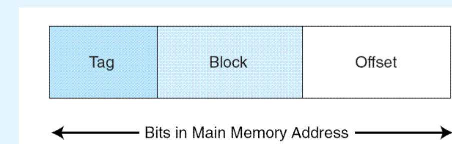
6.4 Cache Memory

- A larger example.



6.4 Cache Memory

- 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

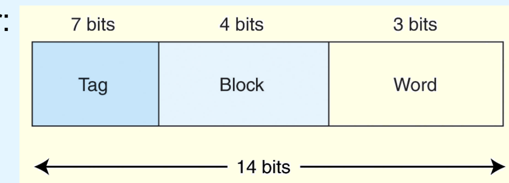
- The diagram below is a schematic of what cache looks like.

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.

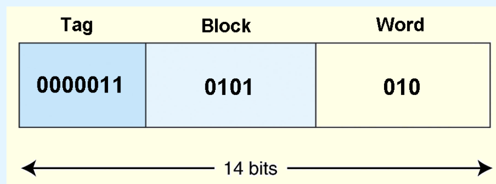
6.4 Cache Memory

- 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 $2^{14} / 2^3 = 2^{11}$ blocks.
- For our field sizes, we know we need 4 bits for the block, 3 bits for the word (offset), and the tag is what's left over:



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**	**011**

- 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 block associated with the **`3AB`** reference.

6.4 Cache Memory

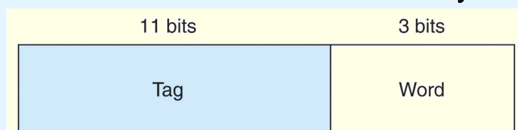
- Suppose a program generates a series of memory references such as: **`1AB, 3AB, 1AB, 3AB`**, . . . The cache will continually evict and replace blocks.
- The theoretical advantage offered by the cache is lost in this extreme case.
- This is the main disadvantage of direct mapped cache.
- Other cache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

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

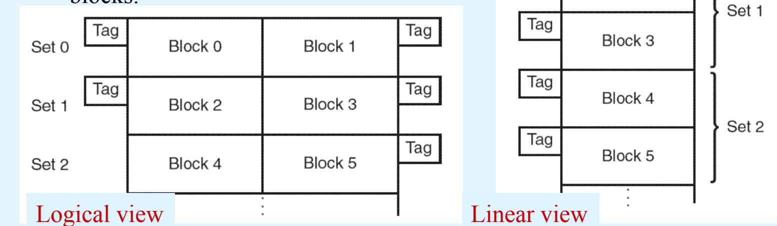
- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the *victim block*.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

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 (*N*) 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.

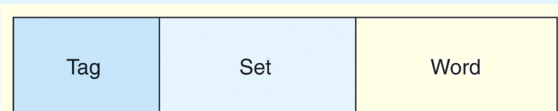
6.4 Cache Memory

- 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.



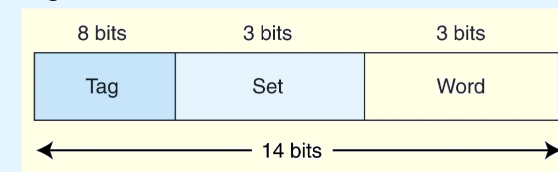
6.4 Cache Memory

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



6.4 Cache Memory

- Set associative cache is a good compromise between direct mapped and fully associative cache.
- Studies indicate that it exhibits good performance, and that 2-way up to 16-way caches perform almost as well as fully associative cache.
- Therefore, most modern computers use some form of set associative cache, with 4-way set associative being one of the most common.

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 accessed 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, but it never thrashes.

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:

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

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

- 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:

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

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%.
- If the accesses do not overlap, the EAT is:

$$0.99(10ns) + 0.01(10ns + 200ns)$$
$$= 9.9ns + 2.01ns = 12ns.$$

- 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

- Caching depends 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 actually cause performance degradation rather than performance improvement.

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

6.4 Cache Memory

- 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 the majority of accesses tend to be reads, not writes.
- Write back (also called *copyback*) updates memory only when the block is selected for replacement.
- 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

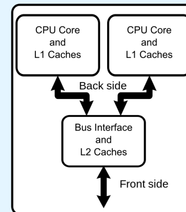
- The cache we have been discussing is called a *unified* or *integrated* cache where both instructions and data are cached.
- Many modern systems employ separate caches for data and instructions.
  - This is called a *Harvard cache*.
- The separation of data from instructions provides better locality, at the cost of greater complexity.
  - Simply making the cache larger provides about the same performance improvement without the complexity.

6.4 Cache Memory

- 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 really contiguous.

6.4 Cache Memory

- Most of today's small systems employ multilevel cache hierarchies.
- The levels of cache form their own small memory hierarchy.
- Level 1 (L1) cache (8KB to 64KB) is situated on the processor itself.
  - Access time is typically about 4ns.
- Level 2 (L2) 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

- 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 (L3) 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

- If the cache system uses an *inclusive* cache, the same data may be present at multiple levels of cache. In the Intel Pentium family, data found in L1 may also be found in L2.
- *Strictly inclusive* caches guarantee that all data at one level is also found in the next lower 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.

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.
- 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

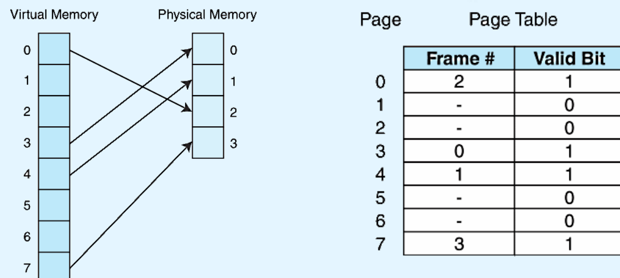
- A *physical address* is the actual memory address of physical memory.
- Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager.
- *Page faults* occur when a logical address requires that a page be brought in from disk.
- *Memory fragmentation* occurs when 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

- 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.



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

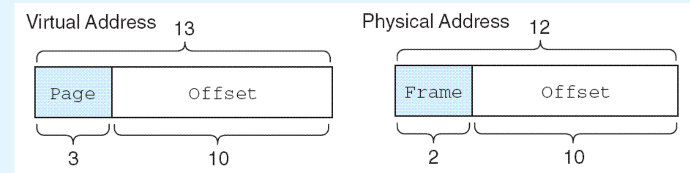
- When a process generates a virtual address, the operating system translates it into a physical memory address.
- To accomplish this, the virtual address is divided into two fields: A *page* field, and an *offset* field.
- The page field determines the page location of the address, and the offset indicates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

- As an example, suppose a system has a virtual address space of 8KB, a physical memory size of 4KB, and a page size of 1KB.
  - 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 for the offset.



6.5 Virtual Memory

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

Page Table

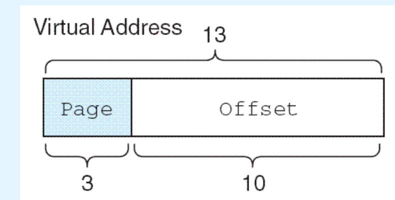
Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Addresses

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

6.5 Virtual Memory

- What happens when 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

- The address 1010101010011_2 is converted to physical address 010101010011_2 = 0x1553 because the page field 101 is replaced by frame number 01

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

Addresses

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

6.5 Virtual Memory

- What happens when the CPU generates address 1000000000100_2?

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

Addresses

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

6.5 Virtual Memory

- 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

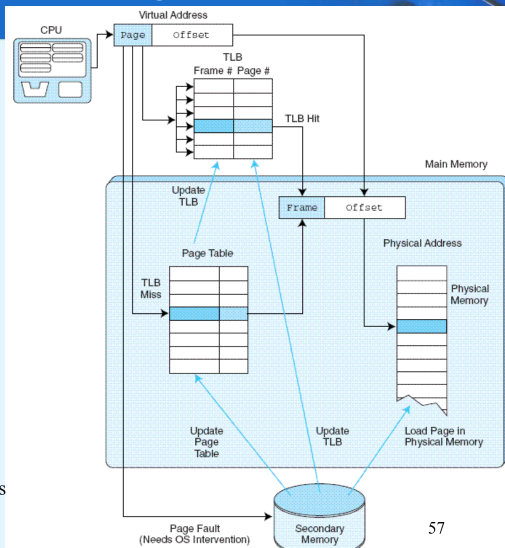
- Even if we had no page faults, the EAT would be 400ns because memory is always read twice: First to access the page table, and second to load the page from memory.
- Because page tables are read constantly, it makes sense to keep them in a special cache called a *translation look-aside buffer* (TLB).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

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

6.5 Virtual Memory

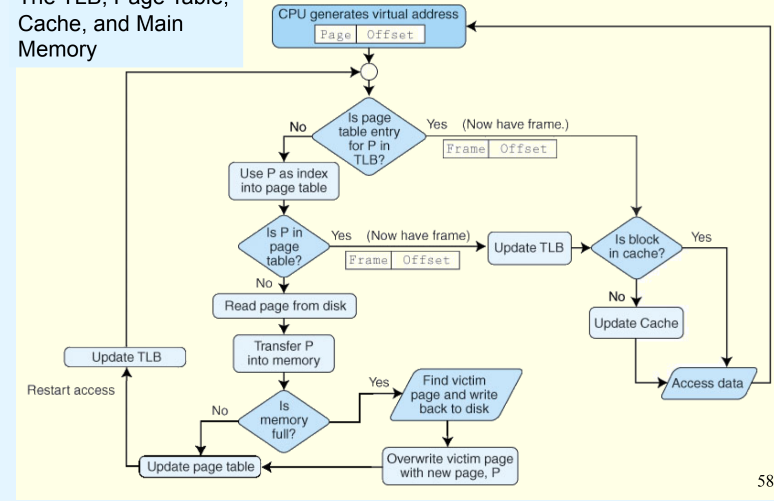
TLB lookup process

1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Search for the virtual page number in the TLB.
4. If the (virtual page #, page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
5. If there is a TLB miss, go to the page table to get the necessary frame number. If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.
6. If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.



6.5 Virtual Memory

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



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.

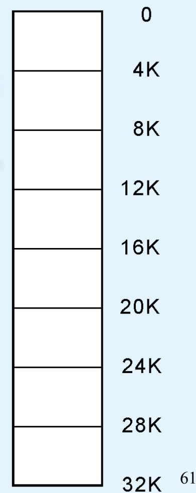
6.5 Virtual Memory

- 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.

The next slides illustrate internal and external fragmentation.

6.5 Virtual Memory

- 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

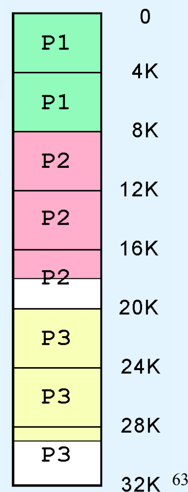
- Suppose there are four processes waiting to be loaded into the system with memory requirements as shown in the table.
- We observe that these processes require 31K of memory.

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

6.5 Virtual Memory

- When the first three processes are loaded, memory looks like this:
- All of the frames are occupied by three of the processes.

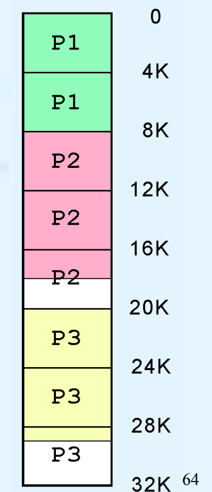
P1	8K
P2	10K
P3	9K
P4	4K



6.5 Virtual Memory

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

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



6.5 Virtual Memory

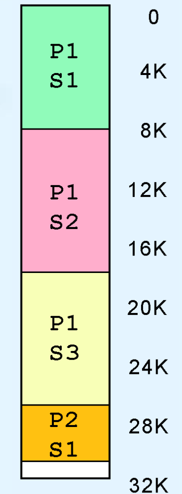
- Suppose that instead of frames, our 32K system uses segmentation.
- The memory segments of two processes is shown in the table at the right.
- The segments can be allocated anywhere in memory.

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

6.5 Virtual Memory

- All of 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.

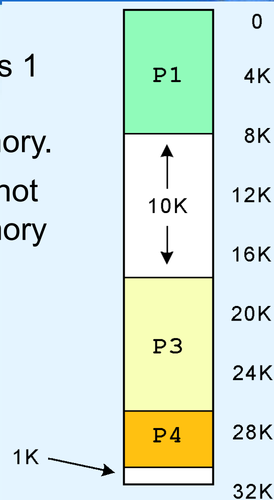
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



6.5 Virtual Memory

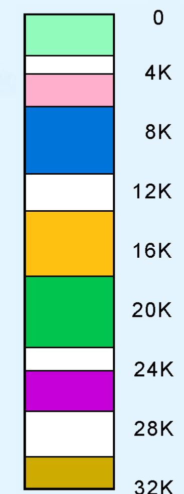
- Eventually, Segment 2 of Process 1 is no longer needed, so it is unloaded giving 11K of free memory.
- But Segment 2 of Process 2 cannot be loaded because the free memory is not contiguous.

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



6.5 Virtual Memory

- 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

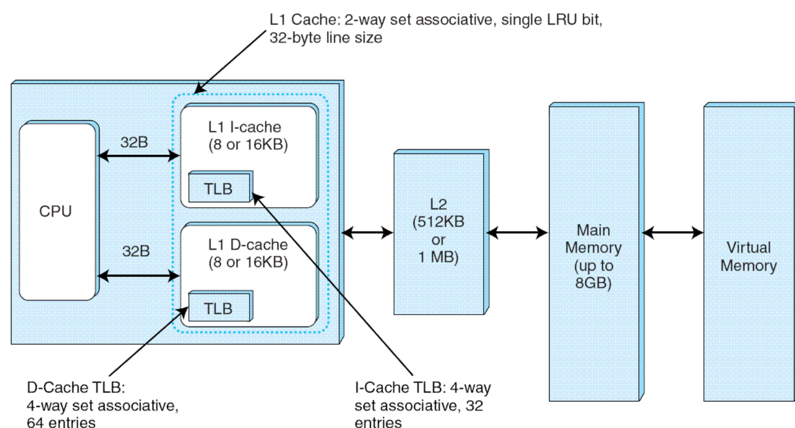
- Large page tables are cumbersome and slow, but with its uniform memory mapping, page operations are fast. Segmentation allows fast access to the segment table, but segment loading is labor-intensive.
- Paging and segmentation can be combined to take advantage of the best features of both by assigning 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: an 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.

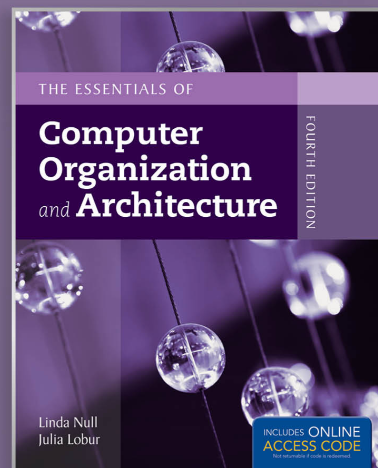
Chapter 6 Conclusion

- 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.

End of Chapter 6

Chapter 7

Input/Output and Storage Systems



© Odua Images/ShutterStock, Inc. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com

Chapter 7 Objectives

- Understand how I/O systems work, including I/O methods and architectures.

7.1 Introduction

- Data storage and retrieval is one of the primary functions of computer systems.
  - One could easily make the argument that computers are more useful to us as data storage and retrieval devices than they are as computational machines.
- All computers have I/O devices connected to them, and to achieve good performance I/O should be kept to a minimum!

7.2 I/O and Performance

- Sluggish I/O throughput can have a ripple effect, dragging down overall system performance.
  - This is especially true when virtual memory is involved.
- The fastest processor in the world is of little use if it spends most of its time waiting for data.
- If we really understand what's happening in a computer system we can make the best possible use of its resources.

7.3 Amdahl's Law

- The overall performance of a system is a result of the interaction of all of its components.
- System performance is most effectively improved when the performance of the most heavily used components is improved.
- This idea is quantified by Amdahl's Law:

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

where S is the overall speedup; f is the fraction of work performed by a faster component; and k is the speedup of the faster component.

7.3 Amdahl's Law

- Amdahl's Law gives us a handy way to estimate the performance improvement we can expect when we upgrade a system component.
- On a large system, suppose we can upgrade a CPU to make it 50% faster for $10,000 or upgrade its disk drives for $7,000 to make them 150% faster.
- Processes spend 70% of their time running in the CPU and 30% of their time waiting for disk service.
- An upgrade of which component would offer the greater benefit for the lesser cost?

7.3 Amdahl's Law

- The processor option offers a 30% speedup:

$$f = 0.70, \quad k = 1.5 \qquad S = \frac{1}{(1 - 0.7) + 0.7/1.5}$$

- And the disk drive option gives a 22% speedup:

$$f = 0.30, \quad k = 2.5 \qquad S = \frac{1}{(1 - 0.3) + 0.3/2.5}$$

- Each 1% of improvement for the processor costs \$10,000/30 = \$333, and for the disk a 1% improvement costs \$7,000/22 = \$318.

Should price/performance be your only concern?

7.4 I/O Architectures

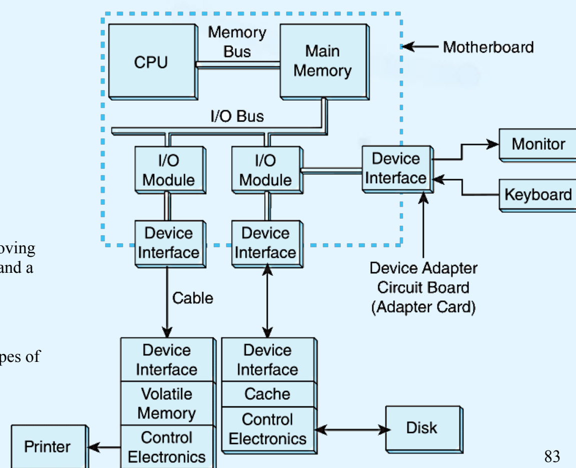
- We define input/output as a subsystem of components that moves coded data between external devices and a host system.
- I/O subsystems include:
  - Blocks of main memory that are devoted to I/O functions.
  - Buses that move data into and out of the system.
  - Control modules in the host and in peripheral devices
  - Interfaces to external components such as keyboards and disks.
  - Cabling or communication links between the host system and its peripherals.

7.4 I/O Architectures

This is a model I/O configuration.

I/O modules takes care of moving data between main memory and a particular device interface.

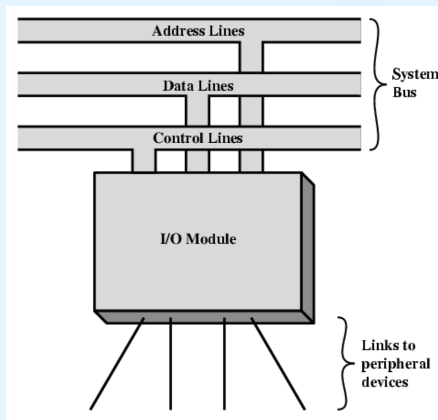
Interfaces are designed to communicate with certain types of devices.

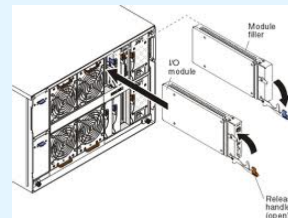


7.4 I/O Architectures

- I/O devices are very different (i.e., keyboard and hard disk performs totally different functions), yet they are both part of the I/O subsystem. All are slower than CPU and RAM.
- The interfaces between the CPU and I/O devices are very similar.
- Each I/O device needs to be connected to:
  - Address bus – to pass address to the peripheral
  - Data bus – to pass data to and from the peripheral
  - Control bus – to control signals to peripherals

7.4 I/O Architectures





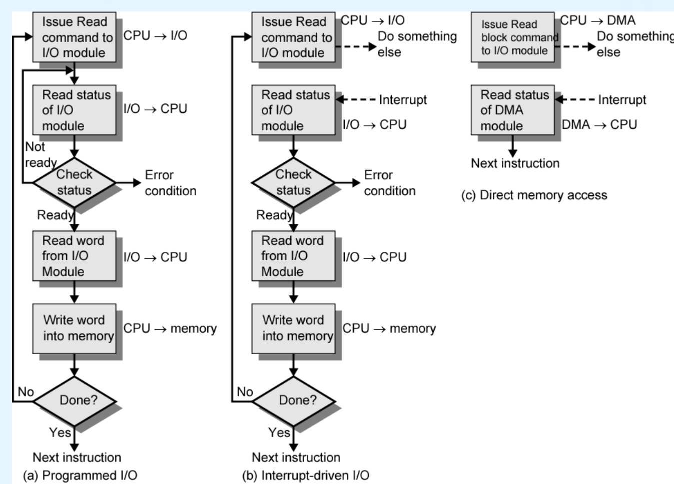
7.4 I/O Architectures

I/O can be controlled in five general ways.

- Programmed I/O reserves a register for each I/O device. Each register is continually polled to detect data arrival. The CPU is busy-waiting.
- Interrupt-Driven I/O allows the CPU to do other things until I/O is requested.
- Memory-Mapped I/O shares memory address space between I/O devices and program memory.
- Direct Memory Access (DMA) offloads I/O processing to a special-purpose chip that takes care of the details.
- Channel I/O uses dedicated I/O processors.

Polling: actively sampling the status of an external device

7.4 I/O Architectures



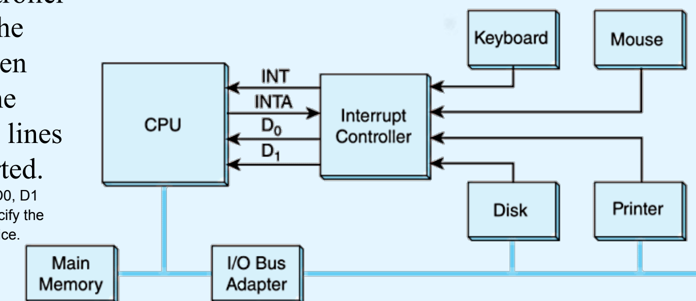
7.4 I/O Architectures

This is an idealized I/O subsystem that uses interrupts.

Each device connects its interrupt line to the interrupt controller.

The controller signals the CPU when any of the interrupt lines are asserted.

The data lines D0, D1 are used to specify the interrupting device.

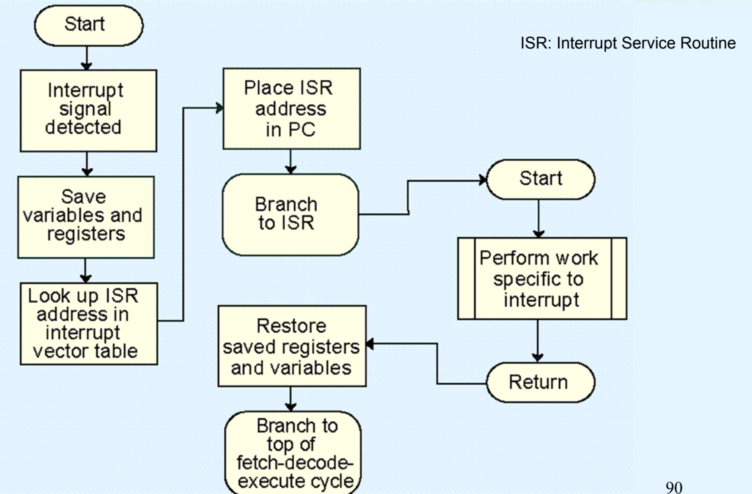


7.4 I/O Architectures

- Recall from Chapter 4 that in a system that uses interrupts, the status of the interrupt signal is checked at the top of the fetch-decode-execute cycle.
- The particular code that is executed whenever an interrupt occurs is determined by a set of addresses called *interrupt vectors* that are stored in low memory.
- The system state is saved before the interrupt service routine is executed and is restored afterward.

We provide a flowchart on the next slide.

7.4 I/O Architectures



7.4 I/O Architectures

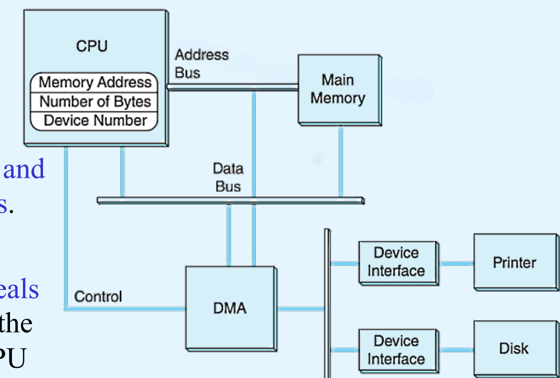
- In memory-mapped I/O devices and main memory share the same address space.
  - Each I/O device has its own reserved block of memory.
  - Memory-mapped I/O therefore looks just like a memory access from the point of view of the CPU.
  - Thus the same instructions to move data to and from both I/O and memory, greatly simplifying system design.
- In small systems the low-level details of the data transfers are offloaded to the I/O controllers built into the I/O devices.

7.4 I/O Architectures

This is a DMA configuration.

Notice that the DMA and the CPU share the bus.

The DMA runs at a higher priority and steals memory cycles from the CPU. Slows down CPU but not as much as CPU doing transfer.



DMA: Direct Memory Access

7.4 I/O Architectures

- A DMA device controller transfers a large block of data directly into or from main memory.
- CPU initiates the transfer by commanding the DMA device to transfer the data and then continue its work.
- The DMA device performs the data transfer and notifies (interrupts) the CPU when it is completed.

- DMA controller
  - a word-count register
  - an address register
  - a data buffer

7.4 I/O Architectures

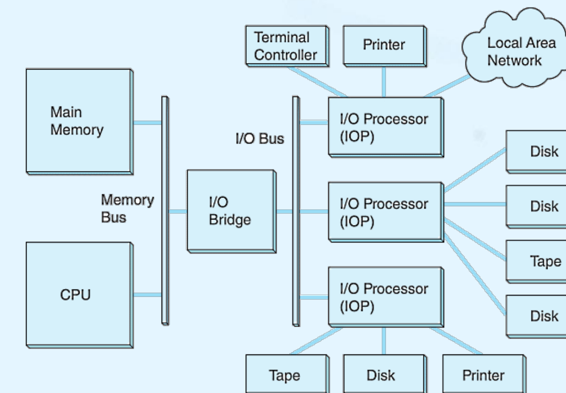
- Very large systems employ channel I/O.
- Channel I/O consists of one or more I/O processors (IOPs) that control various channel paths.
- Slower devices such as terminals and printers are combined (*multiplexed*) into a single faster channel.
- On IBM mainframes, multiplexed channels are called *multiplexor channels*, the faster ones are called *selector channels*.

7.4 I/O Architectures

- Channel I/O is distinguished from DMA by the intelligence of the IOPs.
- The IOP negotiates protocols, issues device commands, translates storage coding to memory coding, and can transfer entire files or groups of files independent of the host CPU.
- The host has only to create the program instructions for the I/O operation and tell the IOP where to find them.

7.4 I/O Architectures

- This is a channel I/O configuration.



7.4 I/O Architectures

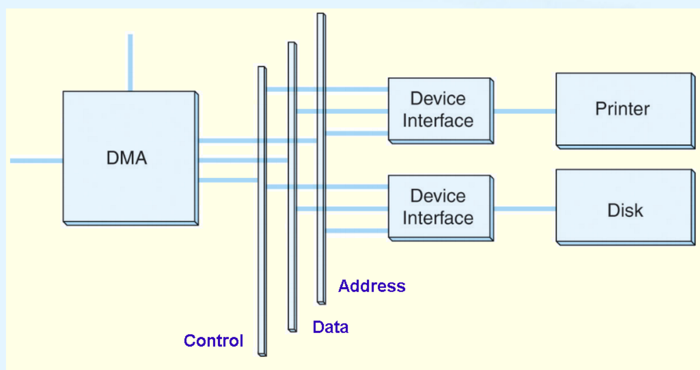
- Character I/O devices process one byte (or character) at a time.
  - Examples include modems, keyboards, and mice.
  - Keyboards are usually connected through an interrupt-driven I/O system.
- Block I/O devices handle bytes in groups.
  - Most mass storage devices (disk and tape) are block I/O devices.
  - Block I/O systems are most efficiently connected through DMA or channel I/O.

7.4 I/O Architectures

- I/O buses, unlike memory buses, operate asynchronously. Requests for bus access must be arbitrated among the devices involved.
- Bus control lines activate the devices when they are needed, raise signals when errors have occurred, and reset devices when necessary.
- The number of data lines is the *width* of the bus.
- A bus clock is required to define bit cell boundaries.

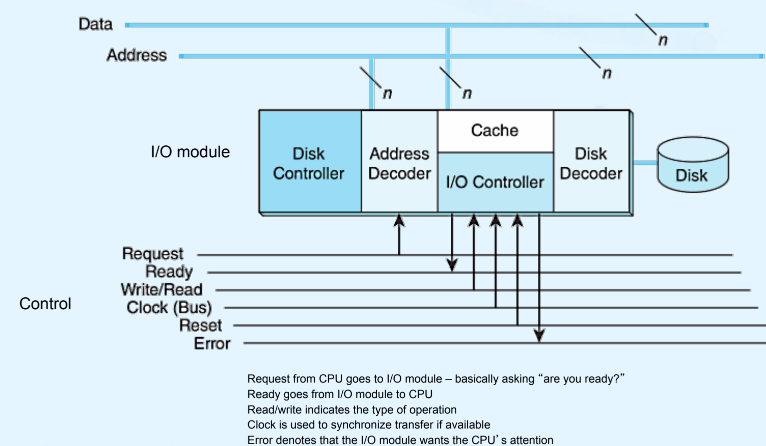
7.4 I/O Architectures

This is a generic DMA configuration showing how the DMA circuit connects to a data bus.



7.4 I/O Architectures

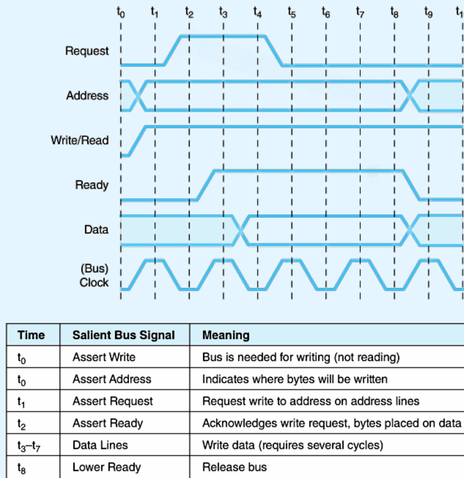
This is how a bus connects to a disk drive.



Request from CPU goes to I/O module – basically asking "are you ready?"
Ready goes from I/O module to CPU
Read/write indicates the type of operation
Clock is used to synchronize transfer if available
Error denotes that the I/O module wants the CPU's attention

7.4 I/O Architectures

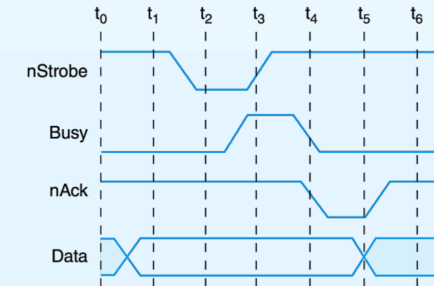
Timing diagrams, such as this one, define bus operation in detail.



Time	Salient Bus Signal	Meaning
t_0	Assert Write	Bus is needed for writing (not reading)
t_0	Assert Address	Indicates where bytes will be written
t_1	Assert Request	Request write to address on address lines
t_2	Assert Ready	Acknowledges write request, bytes placed on data lines
t_3–t_7	Data Lines	Write data (requires several cycles)
t_8	Lower Ready	Release bus

7.5 Data Transmission Modes

- Bytes can be conveyed from one point to another by sending their encoding signals simultaneously using *parallel data transmission* or by sending them one bit at a time in *serial data transmission*.
  – Parallel data transmission for a printer resembles the signal protocol of a memory bus.



nStrobe and nAck are strobe and acknowledgement signals that are asserted when they carry low voltage. The Busy and Data signals are asserted when they carry high voltage.

7.5 Data Transmission Modes

- In parallel data transmission, the interface requires one conductor for each bit.
- Parallel cables are fatter than serial cables.
- Compared with parallel data interfaces, serial communications interfaces:
  – Require fewer conductors.
  – Are less susceptible to attenuation (signal loss over time).
  – Can transmit data farther and faster.

Serial communications interfaces are suitable for time-sensitive (*isochronous*) data such as voice and video.

7.10 The Future of Data Storage

- Advances in technology have defied all efforts to define the ultimate upper limit for magnetic disk storage.
  – In the 1970s, the upper limit was thought to be around $2MB/in^2$. (1 in = 2.54 cm)
  – Today's disks commonly support $20GB/in^2$.
- Improvements have occurred in several different technologies including:
  – Materials science
  – Magneto-optical recording heads
  – Error correcting codes

7.10 The Future of Data Storage

- As data densities increase, bit cells consist of proportionately fewer magnetic grains.
- There is a point at which there are too few grains to hold a value, and a 1 might spontaneously change to a 0, or vice versa.
- This point is called the superparamagnetic limit.
  - In 2006, the superparamagnetic limit is thought to lie between $150GB/in^2$ and $200GB/in^2$.
- Even if this limit is wrong by a few orders of magnitude, the greatest gains in magnetic storage have probably already been realized.

A **magnetic grain** is a unit of material that can be magnetized in a predictable direction. Today, one bit of information, 0 or 1, may require 50-100 grains clumped together for successful storage.

7.10 The Future of Data Storage

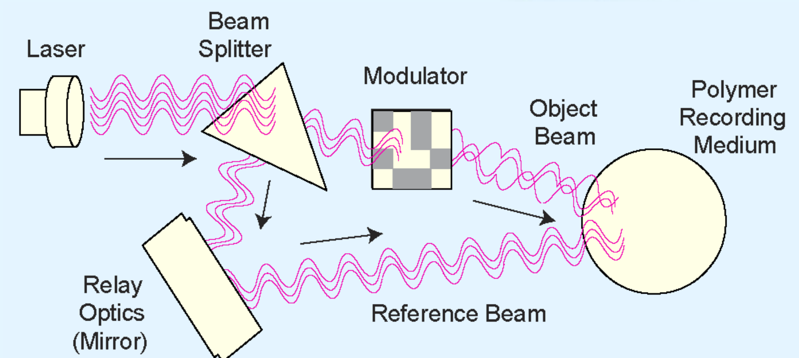
- Future exponential gains in data storage most likely will occur through the use of totally new technologies.
- Research into finding suitable replacements for magnetic disks is taking place on several fronts.
- Some of the more interesting technologies include:
  - Biological materials
  - Holographic systems
  - Micro-electro-mechanical devices.
  - Carbon nanotubes
  - Memristors

7.10 The Future of Data Storage

- Present day biological data storage systems combine organic compounds such as proteins or oils with inorganic (magnetizable) substances.
- Early prototypes have encouraged the expectation that densities of $1TB/in^2$ are attainable.
- Of course, the ultimate biological data storage medium is DNA.
  - Trillions of messages can be stored in a tiny strand of DNA.
- Practical DNA-based data storage is most likely decades away.

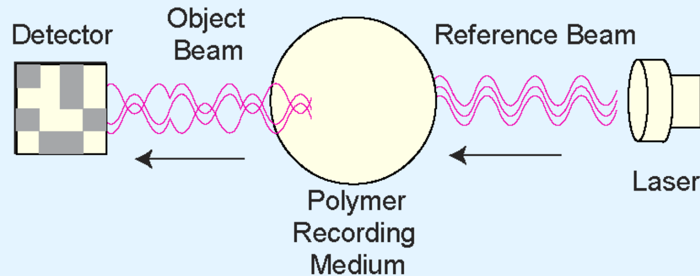
7.10 The Future of Data Storage

- Holographic storage uses a pair of laser beams to etch a three-dimensional hologram onto a polymer medium.



7.10 The Future of Data Storage

- Data is retrieved by passing the reference beam through the hologram, thereby reproducing the original coded object beam.



7.10 The Future of Data Storage

- Because holograms are three-dimensional, tremendous data densities are possible.
- Experimental systems have achieved over 30GB/in^2, with transfer rates of around 1GBps.
- In addition, holographic storage is content addressable.
  - This means that there is no need for a file directory on the disk. Accordingly, access time is reduced.
- The major challenge is in finding an inexpensive, stable, rewriteable holographic medium.

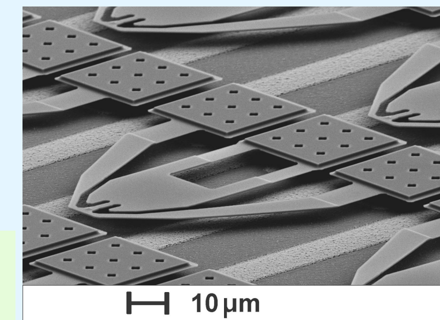
7.10 The Future of Data Storage

- Micro-electro-mechanical storage (MEMS) devices offer another promising approach to mass storage.
- IBM's Millipede is one such device.
- Prototypes have achieved densities of 100GB/in^2 with 1Tb/in^2 expected as the technology is refined.

A photomicrograph of Millipede is shown on the next slide.

7.10 The Future of Data Storage

- Millipede consists of thousands of cantilevers that record a binary 1 by pressing a heated tip into a polymer substrate.
- The tip reads a binary 1 when it dips into the imprint in the polymer

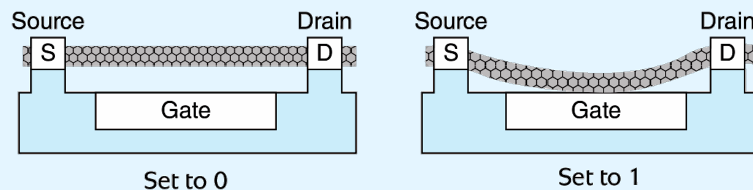


Photomicrograph courtesy of the IBM Corporation.
© 2005 IBM Corporation

A **cantilever** is a beam anchored at only one end

7.10 The Future of Data Storage

- CNTs are a cylindrical form of elemental carbon: The walls of the cylinders are one atom thick.
- CNTs can act like switches, opening and closing to store bits.
- Once "set" the CNT stays in place until a release voltage is applied.



Elemental carbon refers to inorganic forms of carbon

7.10 The Future of Data Storage

- Memristors are electronic components that combine the properties of a resistor with memory.
- Resistance to current flow can be controlled so that states of "high" and "low" store data bits.
- Like CNTs, memristor memories are non-volatile, holding their state until certain threshold voltages are applied.
- These non-volatile memories promise enormous energy savings and increased data access speeds in the very near future.

Chapter 7 Conclusion

- I/O systems are critical to the overall performance of a computer system.
- Amdahl's Law quantifies this assertion.
- I/O systems consist of memory blocks, cabling, control circuitry, interfaces, and media.
- I/O control methods include programmed I/O, interrupt-based I/O, DMA, and channel I/O.
- Buses require control lines, a clock, and data lines. Timing diagrams specify operational details.

Chapter 7 Conclusion

- Any one of several new technologies including biological, holographic, or mechanical may someday replace magnetic disks.
- The hardest part of data storage may be end up be in locating the data after it's stored.

End of Chapter 7