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.

Introduction

- Memory lies at the heart of the stored-program computer (Von Neumann model).
- In previous chapters, we studied the ways in which memory is accessed by various ISAs.
- In this chapter, we focus on memory organization or memory hierarchy systems. 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-only-memory, ROM.
- There are two types of RAM,
  - dynamic RAM (DRAM)
  - 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 Dynamic RAM

- FPM RAM (Fast Page Mode RAM) -- 30 MHz
  - allows faster access to data in the same row or page.
  - works by eliminating the need for a row address if data is located in the row previously accessed.
- EDO RAM (enhanced data-out RAM) -- 66 MHz
  - can start fetching the next block of memory at the same time that it sends the previous block to the CPU
- BEDO RAM (burst enhanced data-out RAM)
  - can process four memory addresses in one *burst*
  - can only stay synchronized with the CPU clock for short periods
  - can't keep up with processors whose buses run faster than 66 MHz

6.2 Dynamic RAM

- SDRAM (synchronous dynamic RAM) -- 100 MHz
  - can run at much higher clock speeds than conventional memory
  - synchronizes itself with the CPU's bus and is capable of running at 133 MHz, about three times faster than conventional FPM RAM, and about twice as fast EDO DRAM and BEDO DRAM
- DDR RAM (double data rate SDRAM) – 200MHz
  - a type of SDRAM that supports data transfers on both edges of each clock cycle (the rising and falling edges), effectively doubling the memory chip's data throughput
  - DDR-SDRAM also consumes less power

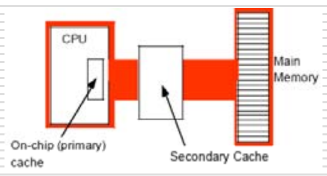
6.2 Static RAM

- SRAM consists of circuits similar to the D flip-flop.
- SRAM is very fast memory and it doesn't need to be refreshed like DRAM does. It is used to build cache memory.
- 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.

6.2 Static RAM

RAM chip primary for special high-speed memory called *level-1 cache* memory

- SRAM (static RAM) --
  - faster and more expensive than DRAM
  - speeds between 8 and 12 ns
  - synchronous or asynchronous
  - does not require a refresh operation
- PBSRAM (pipeline burst SRAM) --
  - collect and send multiple request for memory as a single pipelined request

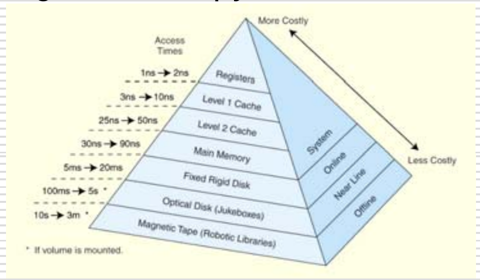


6.3 The Memory Hierarchy

- 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:



Storage Hierarchy

- On-line storage (primary storage):
  - A storage that is actively accessible by the computer without human interaction
  - Hard drive
- Near-on-line storage (secondary storage)
  - A storage that can be accessible by the computer human interaction
  - floppy disk
  - CD-R
- Off-line storage (archival storage)
  - Use as a backup
  - magnetic tapes

6.3 The Memory Hierarchy

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

6.3 The Memory Hierarchy

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

6.3.1 Locality of Reference

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

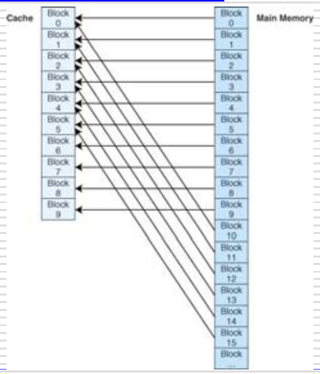
- The "content" that is addressed in content addressable cache memory is a subset of the bits of a main memory address called a field.
- The fields into which a memory address is divided provide a many-to-one mapping between larger main memory and the smaller cache memory.
- Many blocks of main memory map to a single block of cache. A tag field in the cache block distinguishes one cached memory block from another.

6.4.1 Cache Mapping Schemes

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

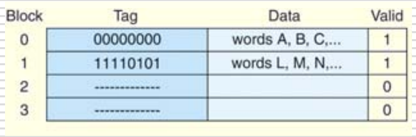
What could happen without having a valid bit?

6.4.1 Direct Mapping Scheme



6.4.1 Direct Mapping Scheme

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

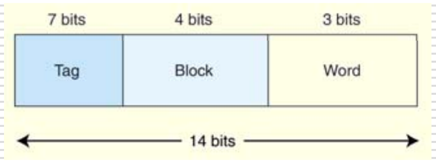


Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	-------------		0
3	-------------		0

- 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.

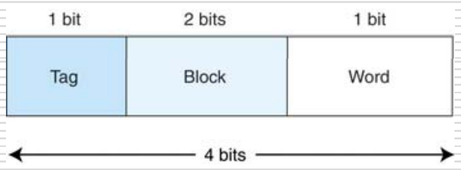
6.4.1 Direct Mapping Scheme

- 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, and the tag is what's left over:



6.4.1 Direct Mapping Scheme

- As an example, suppose a system using direct mapping with 16 words of main memory divided into 8 blocks, 4 blocks cache.
- Main memory consists of 2^4 words, cache has 4 = 2^2 blocks
- For our field sizes, we know we need 2 bits for the block, 1 bit for the word, and 1 bit for the tag

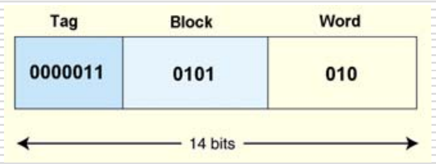


6.4.1 Direct Mapping Scheme

Main Memory	Maps to	Cache
Block 0 (addresses 0, 1)	→	Block 0
Block 1 (addresses 2, 3)	→	Block 1
Block 2 (addresses 4, 5)	→	Block 2
Block 3 (addresses 6, 7)	→	Block 3
Block 4 (addresses 8, 10)	→	Block 0
Block 5 (addresses 10, 11)	→	Block 1
Block 6 (addresses 12, 13)	→	Block 2
Block 7 (addresses 14, 15)	→	Block 3

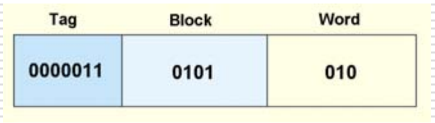
6.4.1 Direct Mapping Scheme

- 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.



6.4.1 Direct Mapping Scheme

- If subsequently the program generates the address 1AB, it will find the data it is looking for in block 0101, word 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 blocks associated with the 3AB reference.

Example

1. Suppose a computer using 15-bit main memory addresses and 64 blocks of cache, each block contains 8 words.
   a) How many blocks of main memory are there?
   b) What is the format of a memory address as seen by the cache, i.e., what are the sizes of the tag, block, and word fields?
   c) To which cache block will the memory reference 1028 map?

Problem 1 on Page 320

1. Suppose a computer using direct mapped cache has 2^{20} words of main memory, and a cache of 32 blocks, where each cache block contains 16 words.
   a) How many blocks of main memory are there?
   b) What is the format of a memory address as seen by the cache, i.e., what are the sizes of the tag, block, and word fields?
   c) To which cache block will the memory reference 0DB6316 map?

6.4.1 Direct Mapping Scheme

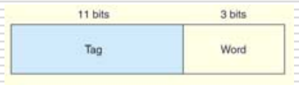
- Suppose a program generates a series of memory references such as: 1AB, 3AB, 1AB, 3AB, . . . The cache will continually evict and replace blocks.
- The theoretical advantage offered by the cache is lost in this extreme case.
- This is the main disadvantage of direct mapped cache.
- Other cache mapping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

6.4.1 Fully Associate Cache

- 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.

6.4.1 Fully Associate Cache

- 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.

Problem 3 on Page 320

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

6.4.1 Cache Mapping Schemes

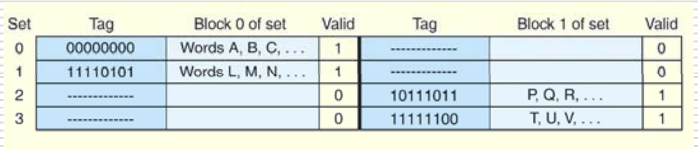
- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the victim block.
-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

6.4.1 Set Associate Cache

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

6.4.1 Set Associate Cache

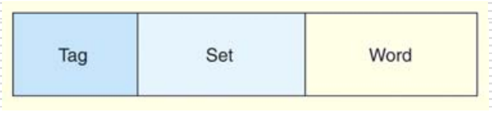
- The number of cache blocks per set in set associative cache varies according to overall system design.
- For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
- Each set contains two different memory blocks.



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

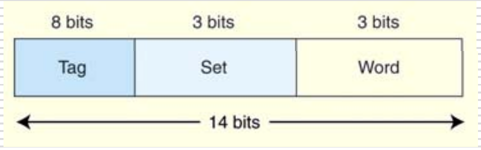
6.4.1 Set Associate Cache

- 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.

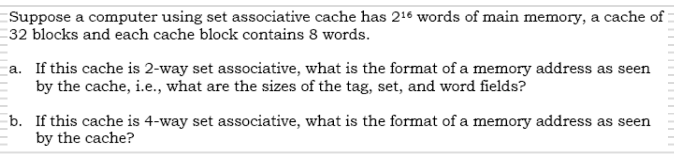


6.4.1 Set Associate Cache

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



Problem 7 Page 321



Suppose a computer using set associative cache has 2^{16} words of main memory, a cache of 32 blocks and each cache block contains 8 words.

a. If this cache is 2-way set associative, what is the format of a memory address as seen by the cache, i.e., what are the sizes of the tag, set, and word fields?

b. If this cache is 4-way set associative, what is the format of a memory address as seen by the cache?

6.4.2 Replacement Policies

- 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.

6.4.2 Replacement Policies

- 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.

6.4.2 Replacement Policies

- 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.

6.4.3 Effective Access Time and Hit Ratio

- 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:

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

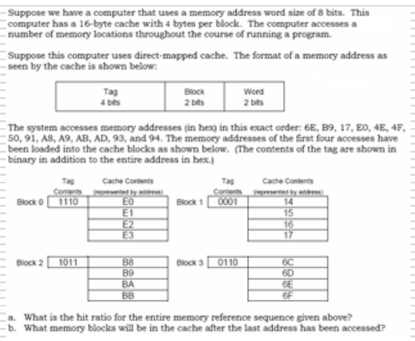
  where H is the cache hit rate and $Access_C$ and $Access_{MM}$ are the access times for cache and main memory, respectively.

6.4.3 Effective Access Time and Hit Ratio

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

  0.99(10ns) + 0.01(200ns) = 9.9ns + 2ns = 11ns.
- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

Problem 9 Page 321



Suppose we have a computer that uses a memory address word size of 8 bits. This computer has a 16-byte cache with 4 bytes per block. The computer accesses a number of memory locations throughout the course of running a program.

Suppose this computer uses direct-mapped cache. The format of a memory address as seen by the cache is shown below:

Tag 4 bits	Block 2 bits	Word 2 bits

The system accesses memory addresses (in hex) in this exact order: 6E, B9, 17, E0, 4E, 4F, 50, 91, A8, A9, AB, AD, 93, and 94. The memory addresses of the first four accesses have been loaded into the cache blocks as shown below. (The contents of the tag are shown in binary in addition to the entire address in hex.)

	Tag Contents	Cache Contents (represented by address)
Block 0	1110	E0
		E1
		E2
		E3
Block 1	0001	14
		15
		16
		17
Block 2	1011	B8
		B9
		BA
		BB
Block 3	0110	6C
		6D
		6E
		6F

a. What is the hit ratio for the entire memory reference sequence given above?
b. What memory blocks will be in the cache after the last address has been accessed?

6.4.5 Cache Write Policies

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

6.4.5 Cache Write Policies

- Write back (also called copyback) updates memory only when the block is selected for replacement.
- The disadvantage of write through is that memory must be updated with each cache write, which slows down the access time on updates. This slowdown is usually negligible, because the majority of 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.

6.5 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

- *Virtual address* -- The logical or program address that the process uses. The CPU generates an address in terms of virtual address.
- *Physical address* -- The real address in physical memory.
- *Mapping* -- The mechanism by which virtual addresses are translated into physical ones.
- *Page frames* -- The equal-size blocks into which main memory is divided.
- *Pages* -- The blocks into which virtual memory is divided, each equal in size to page frame.
- *Paging* -- The process of coping a virtual page from disk to page frame in main memory.
- *Fragmentation* -- Memory that becomes unsuable.
- *Page fault* -- An event that occurs when a requested page is not in main memory and must be copied into memory fromdisk.

6.5.1 Paging

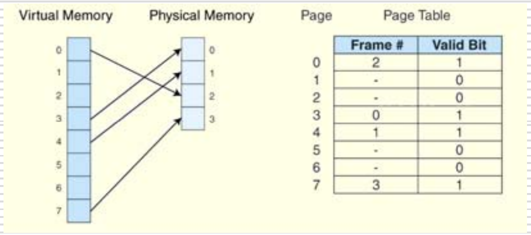
- A *physical address* is the actual memory address of physical memory.
- Programs create *virtual addresses* that are mapped to physical addresses by the memory manager.
- *Page faults* occur when a logical address requires that a page be brought in from disk.
- Memory *fragmentation* occurs when the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

6.5.1 Paging

- 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.

6.5.1 Paging

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

6.5.1 Paging

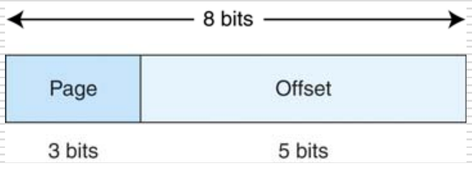
- When a process generates a virtual address, the operating system translates it into a physical memory address.
- To accomplish this, the virtual address is divided into two fields: A page field, and an offset field.
- The page field determines the page location of the address, and the offset indicates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

6.5.1 Paging

- 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.

6.5.1 Paging

- Example: suppose a system has a virtual address space of 2^8 words and a physical address space of 4 page frames of 32 words each frame.
  - We have $2^8/2^5 = 2^3$ virtual pages.
- A virtual address has 8 bits (2^8) with 5 bits for the page field and 3 for the offset.
- A physical memory address requires 7 bits ($2^2 2^5$), the first two bits for the page frame and the trailing 5 bits the offset.

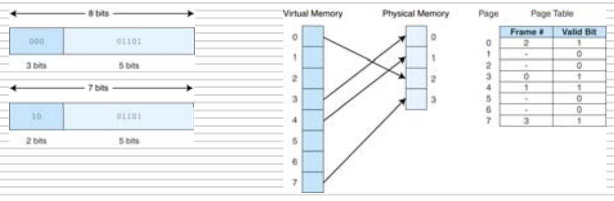


6.5.1 Paging

1. Extract the page number from the virtual address
2. Extract the offset from the virtual address
3. Translate the page number into 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 the page
      - If the valid bit = 0, the system generates a page fault and the operating system must intervene to
        i. Locate the desired page on disk
        ii. Find a free page frame. This may necessitate removing a "victim" page from memory and copying it back to disk if memory is full.
        iii. Copy the desired page into the free page frame in main memory.
        iv. Update the page table. This 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.
        v. Resume execution of the process causing the page fault, cotinuing to step 2.
      - If the valid bit = 1, the page is in memory.
        1. Replace the virtual page number with the actual frame number.
        2. Access data at offset in physical page frame by adding the offset to the frame number for the given virtual page.

6.5.1 Paging

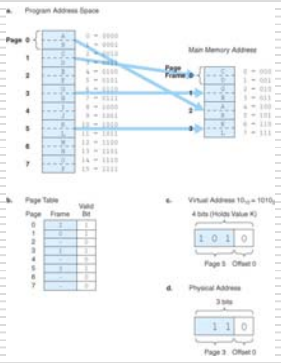
Suppose CPU generates the virtual address
13_{10} = $0D_{16}$ = 00001101_2



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

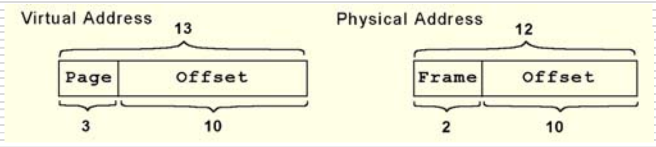
6.5.1 Paging

Example: suppose a program is 16 bytes long, has access to an 8-byte memory that use byte addressing and a page is 2 bytes in length. The program generates the following address reference strings: 0,1,2,3,4,6,7,10,11.



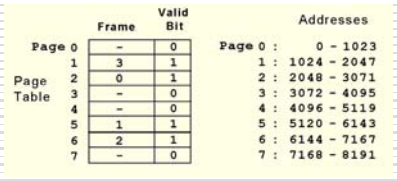
6.5.1 Paging

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



6.5.1 Paging

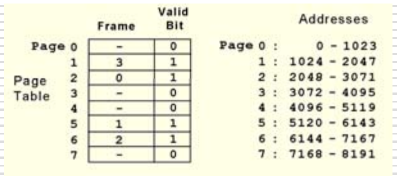
- Suppose we have the page table shown below.
- What happens when CPU generates address 5459_{10} = 1010101010011_2?



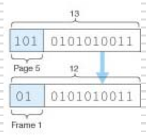
Page Table	Frame	Valid Bit	Addresses
Page 0	–	0	Page 0 : 0 - 1023
1	3	1	1 : 1024 - 2047
2	0	1	2 : 2048 - 3071
3	–	0	3 : 3072 - 4095
4	–	0	4 : 4096 - 5119
5	1	1	5 : 5120 - 6143
6	2	1	6 : 6144 - 7167
7	–	0	7 : 7168 - 8191

6.5.1 Paging

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

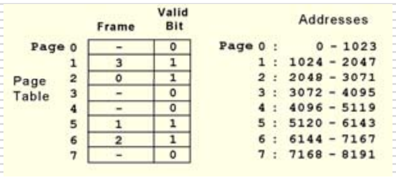


Page Table	Frame	Valid Bit	Addresses
Page 0	–	0	Page 0 : 0 - 1023
1	3	1	1 : 1024 - 2047
2	0	1	2 : 2048 - 3071
3	–	0	3 : 3072 - 4095
4	–	0	4 : 4096 - 5119
5	1	1	5 : 5120 - 6143
6	2	1	6 : 6144 - 7167
7	–	0	7 : 7168 - 8191

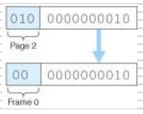


6.5.1 Paging

- The address 2050_{10} = 010000000010_2 is converted to physical address 00000000010 because the page field 010 is replaced by frame number 00 through a lookup in the page table.

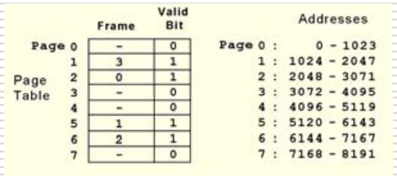


Page Table	Frame	Valid Bit	Addresses
Page 0	–	0	Page 0 : 0 - 1023
1	3	1	1 : 1024 - 2047
2	0	1	2 : 2048 - 3071
3	–	0	3 : 3072 - 4095
4	–	0	4 : 4096 - 5119
5	1	1	5 : 5120 - 6143
6	2	1	6 : 6144 - 7167
7	–	0	7 : 7168 - 8191

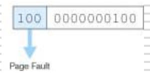


6.5.1 Paging

- What happens when the CPU generates address 1000000000100_2?



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



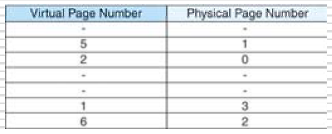
6.5.2 Effective Access Time Using Paging

- 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.

6.5.2 Effective Access Time Using Paging

- 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.

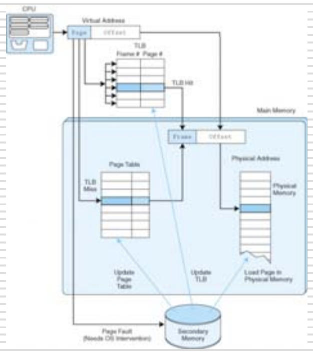


Virtual Page Number	Physical Page Number
-	-
5	1
2	0
-	-
-	-
1	3
6	2

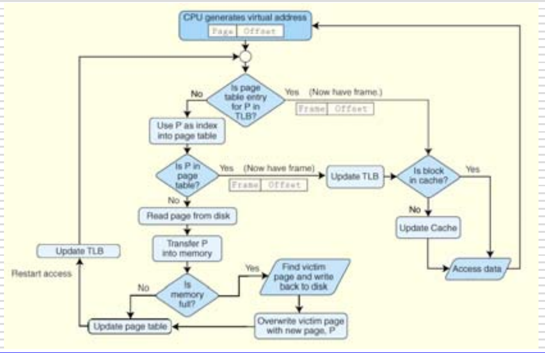
6.5.2 Translation look-aside buffer

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 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.

6.5.2 Translation look-aside buffer



6.5.3 Using Cache, TLBs, and Paging



6.5.4 Pros and Cons of Paging and Virtual Memory

Cons:

- Time for memory referencing
- More time needed if using TLB to cache table entries
- Memory overhead for storing page tables
- Require special hardware and operating system support

Pros:

- Programs are not restricted by the amount of physical memory
- Run more programs

6.5.5 Segmentation

- 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.

6.5.5 Segmentation

- 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.

6.5.6 Paging Combined with Segmentation

- 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.

6.4 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.
- Level1 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

- 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.

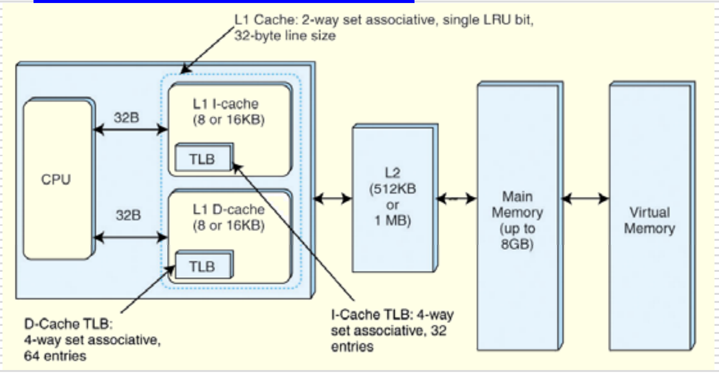
6.4 Cache Memory

- 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.

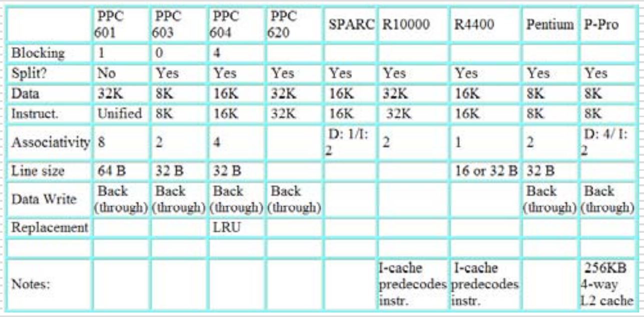
6.6 A Real-World Example

- The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpaged unsegmented, segmented unpaged, and unsegmented paged.
- The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes.
- The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- The L1 cache is in two parts: and instruction cache (I-cache) and a data cache (D-cache).
- MESI cache coherency protocol: Every cache line is marked with one of the four following states:
  - **M** - Modified: The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state.
  - **E** - Exclusive: The cache line is present only in the current cache, but is clean; it matches main memory.
  - **S** - Shared: Indicates that this cache line/block may be stored in other caches of the machine.
  - **I** - Invalid: Indicates that this cache line/block is invalid.

6.6 A Real-World Example


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

6.6 A Real-World Example



	PPC 601	PPC 603	PPC 604	PPC 620	SPARC	R10000	R4400	Pentium	P-Pro
Blocking	1	0	4						
Split?	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data	32K	8K	16K	32K	16K	32K	16K	8K	8K
Instruct.	Unified	8K	16K	32K	16K	32K	16K	8K	8K
Associativity	8	2	4		D: 1/I: 2	2	1	2	D: 4/ I: 2
Line size	64 B	32 B	32 B				16 or 32 B	32 B	
Data Write	Back (through)	Back (through)	Back (through)	Back (through)				Back (through)	Back (through)
Replacement			LRU						
Notes:						I-cache predecodes instr.	I-cache predecodes instr.		256KB 4-way L2 cache

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.

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 (least frequently used). 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.