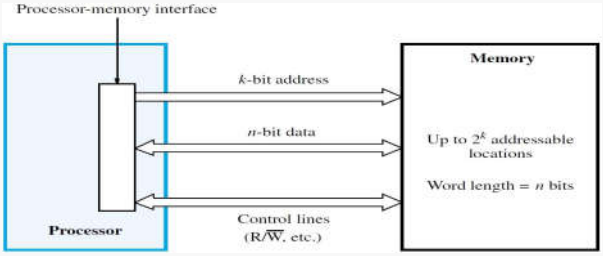
Memory

D.N.Rakhmatov

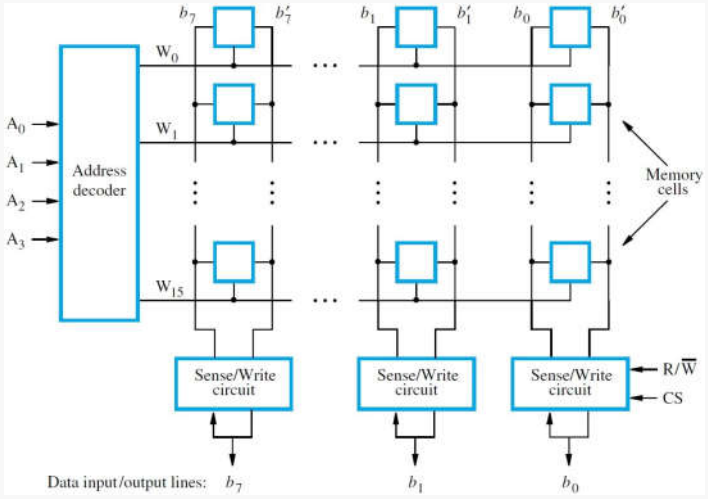
Adopted (with modifications) from:
R. Bryant, CMU
D. Patterson, UC-Berkeley
M. Kowarschik, FAU-Erlangen
W. Stallings, *Operating Systems*, 6/E, © 2009 Pearson
N. Weste, D. Harris, *CMOS VLSI Design*, 4/E, © 2010 Pearson
C. Hamacher et al, *Computer Organization*, 6/E, © 2011 McGraw-Hill
V. Heuring, H. Jordan, *Computer Systems Design*, 2/E, © 2004 Pearson
A. Silberschatz et al, *Operating System Concepts Essentials*, 8/E, © 2011 Wiley
M. Mano, C. Kime, *Logic and Computer Design Fundamentals*, 4/E, © 2008 Pearson
S. Dandamudi, *Fundamentals of Computer Organization and Design*, © 2002 Springer

Generic Memory Setup

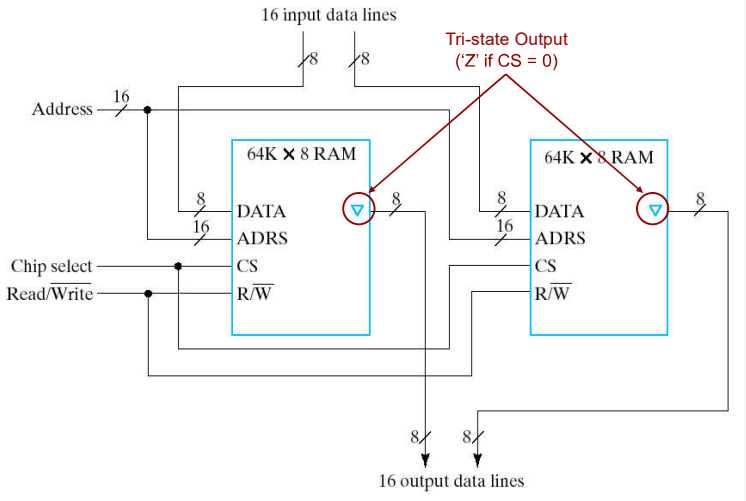


- **Problem:** Main memory is slower than CPU
  - **Solution:** Cache memory (smaller and faster) holds copies of "critical" instructions and data
- **Problem:** Information for one or more active programs may exceed physical memory capacity
  - **Solution:** Virtual memory provides for larger apparent memory size by transparently using secondary storage

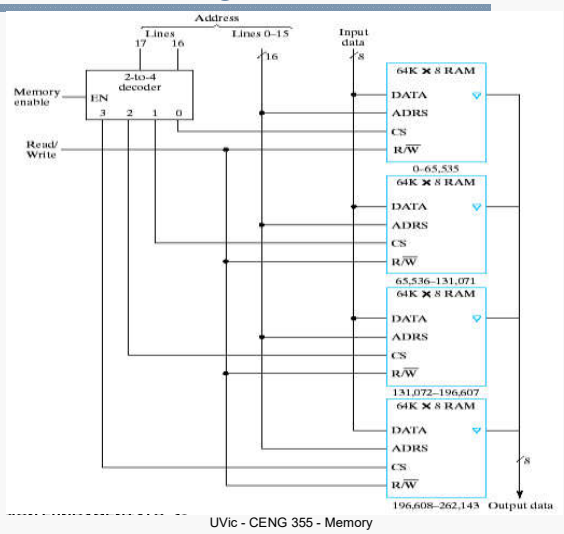
Memory Internal Structure (16×8)



64K×16 Using 64K×8



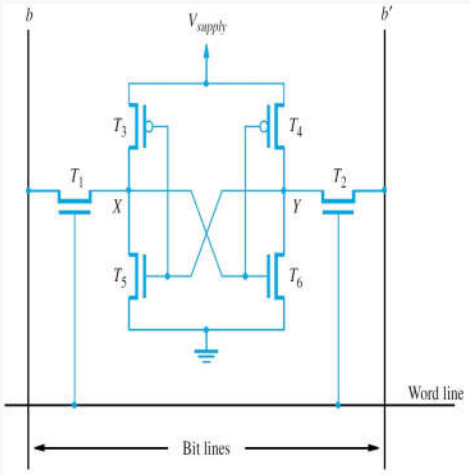
256K×8 Using 64K×8

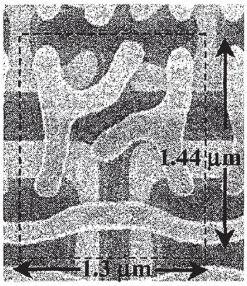


Volatile and Nonvolatile Memories

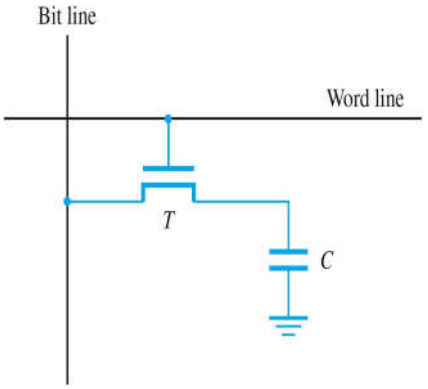
- **Volatile** memories lose information if powered off
  - Generic name is *random-access memory* (RAM)
    - Misleading because some RAMs can hold bits without power
    - **Types:** static RAM (SRAM), dynamic RAM (DRAM), non-volatile RAM (NVRAM, e.g., phase-change RAM)
  - Also important: *content-addressable memory* (CAM)
- **Nonvolatile** memories retain information even when powered off
  - Generic name is *read-only memory* (ROM)
    - Misleading because some ROMs can be reprogrammed
    - **Types:** mask ROM, programmable ROM (PROM), erasable programmable ROM (EPROM), electrically erasable programmable ROM (EEPROM), Flash
  - Firmware = special programs stored in ROM, such as bootstrap code

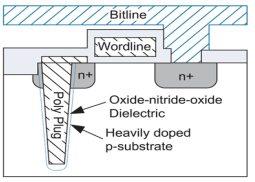
Static Memory Cell (SRAM)

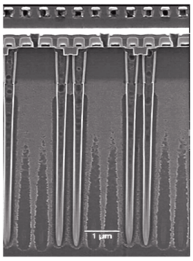




Dynamic Memory Cell (DRAM)



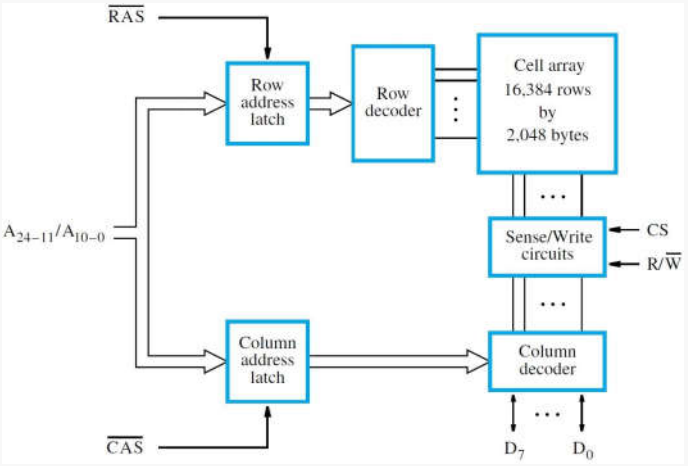




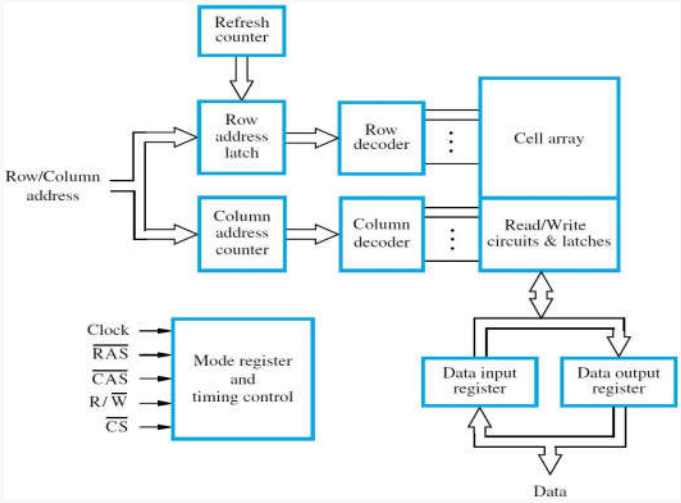
SRAM vs DRAM

- **SRAM:** Static RAM (used for caches)
  - Cell uses **6** transistors to store **1** bit and holds data as long as power is supplied
  - Faster, but more expensive than denser DRAM
- **DRAM:** Dynamic RAM (used for main memory)
  - Cell uses **1** transistor and **1** capacitor to store **1** bit
  - "**Refresh**" required due to charge leakage
    - Word's cells refreshed when read
    - Refresh circuitry strobes consecutive memory addresses periodically, causing memory content to be refreshed
  - Address bus multiplexed between row and column components
    - Row and column addresses are latched in sequentially, using RAS (row address strobe) and CAS (column address strobe) signals

Asynchronous DRAM (32M×8)



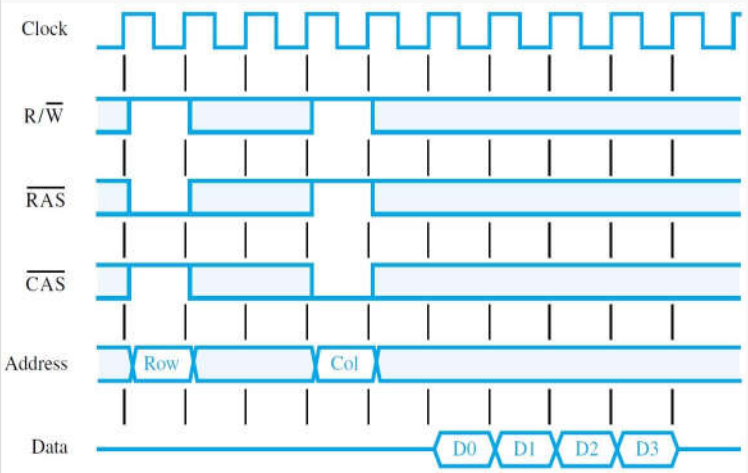
Synchronous DRAM (SDRAM)



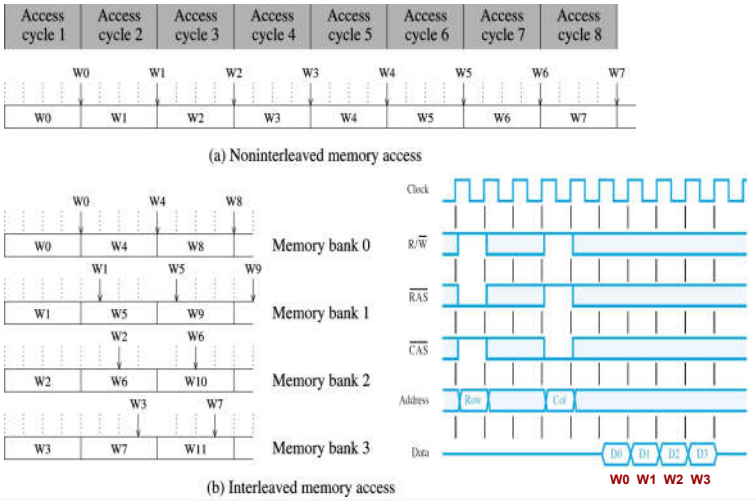
More on SDRAM

- SDRAMs include data registers, in addition to address latches
  - New access operation can be initiated while data are transferred to/from these data registers
- SDRAM have more sophisticated control circuitry
  - E.g., built-in refresh circuit with refresh counter
  - E.g., column address counter for data block transfers
- SDRAMs require power-up configuration
  - The memory controller needs to initialize SDRAM's mode register used to specify the *burst length* for block transfers and to set delays for timing control
- Double-data rate (DDR) SDRAMs
  - Use both rising and falling clock edges after RAS/CAS assertion and interleave consecutive data across two SDRAM arrays (switching between the arrays at each clock edge)

SDRAM Read Example



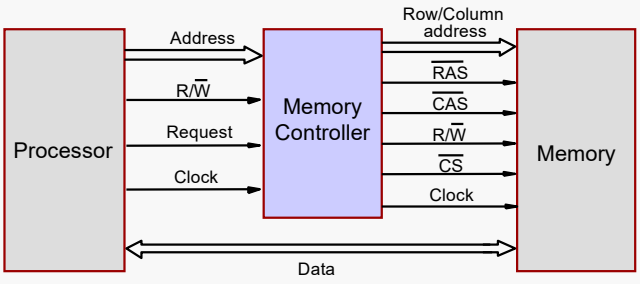
Interleaving Example



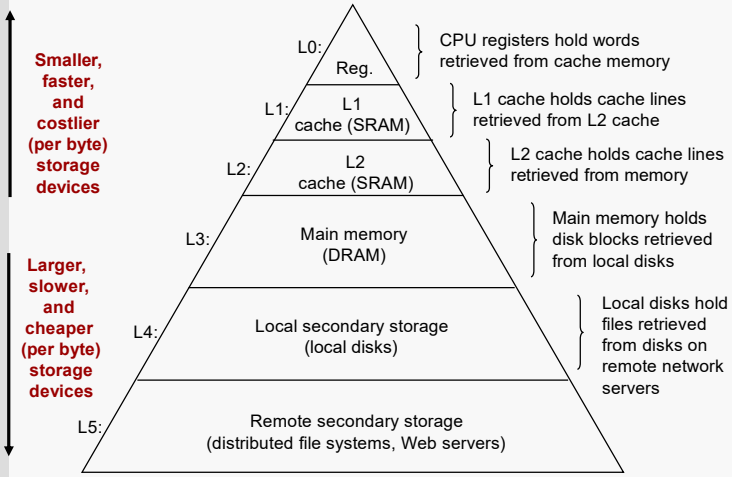
DRAM Refresh

- DRAM cells must be periodically **refreshed** to compensate for charge leakage
  - Refresh is either built-in into a DRAM itself or managed by the memory controller
  - Typical refresh period: ~ tens of ms
- Example:
  - **8K** rows with **4**-cycle access, **133** MHz clock
  - **8,192** x **4** = **32,768** cycles to refresh all rows
  - Refresh time **32,768** / **133**×**10**$^{-6}$ = **246**×**10**$^{-6}$ seconds
  - If refresh period is **64** ms, then actual refresh overhead is **0.246** / **64** = 0.4%

Memory Controller



Memory Hierarchy



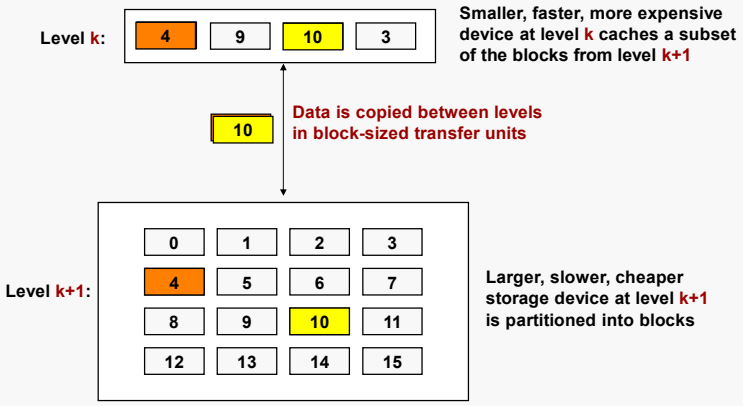
Why Memory Hierarchy?

- Properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity
  - The CPU-memory speed gap is widening
  - Well-written programs tend to exhibit locality
    - *"A program spends **90%** of its time in **10%** of its code"*
- These fundamental properties nicely complement each other
  - Lucky for us!
  - We need to organize memory and storage systems in an hierarchical fashion

Caches

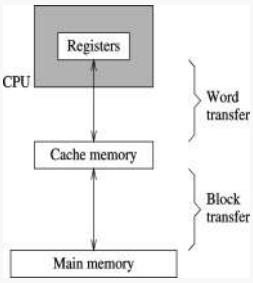
- **Cache:**
  - A smaller, faster storage device that acts as a staging area for a subset of instructions and data in a larger, slower device
- Fundamental idea of a memory hierarchy:
  - For each **k**, the faster, smaller device at level **k** serves as a cache for the larger, slower device at level **k+1**
  - Why do memory hierarchies work?
    - Programs tend to access instructions and data at level **k** more often than they access them at level **k+1**
    - Thus, the storage at level **k+1** can be slower, and thus larger and cheaper per bit
- Net effect: a large storage that costs as low as the cheap memory, but serves instructions and data to processors as quickly as the fast memory

Caching Concept



Cache Terminology

- Cache hit or miss
  - Successful or failed access to a word in a cache
- Cache line (block)
  - Several words residing at adjacent addresses that are cached all together upon a miss
- Write policies
  - **Write-through:** cache location and main memory location are updated simultaneously
  - **Write-back:** main memory location is updated later, only if a special hardware flag bit ("dirty", or "modified") is set to indicate the associated cache word has been changed



Cache Performance Metrics

- Processor looks first for data in the primary cache (L1), then in the secondary cache (L2), then in main memory
- **Miss rate:** fraction of memory references not found in cache
  - Typically, **3-10%** for L1, can be **< 1%** for L2
- **Hit time:** time to deliver a word in the cache to the processor (including time to determine if the word is in the cache)
  - Typically, **1** clock cycle for L1, **3-8** clock cycles for L2
- **Miss penalty:** additional time required because of a miss
  - Typically, **25-100** cycles for main memory

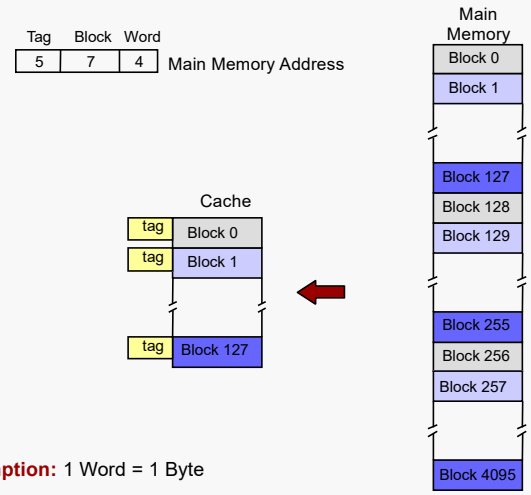
Cache Benefit

- Average access time with cache:
  - $T_{ave} = hC + (1 - h)M$
  - Example:
    - **10**τ seconds for each memory read without cache
    - **1**τ seconds cache hit time, **19**τ seconds cache miss penalty
    - **130** memory accesses per **100** instructions
    - Instruction hit rate = **95%**, data hit rate = **90%**
    - Instruction miss rate = **5%**, data miss rate = **10%**
    - Performance improvement:
      - ✓ $130 \times 10\tau / (100(0.95 \times 1\tau + 0.05 \times 19\tau) + 30(0.9 \times 1\tau + 0.1 \times 19\tau)) = 4.7$
    - If the hit rate is **100%**, then additional improvement:
      - ✓ $(100(0.95 \times 1\tau + 0.05 \times 19\tau) + 30(0.9 \times 1\tau + 0.1 \times 19\tau)) / 130 = 2.1$
- Two levels of caching:
  - $T_{ave} = h_1C_1 + (1 - h_1)h_2C_2 + (1 - h_1)(1 - h_2)M$

Replacement Policy

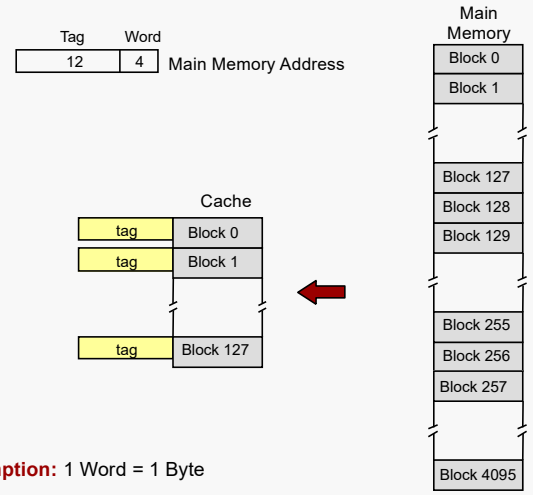
- When the cache is full and a word that is not in the cache is referenced, the cache control hardware follows some rule to decide which block should be removed to create space for new block containing the referenced word
  - Objective: keep blocks that are likely to be referenced in the near future
- **LRU** policy: the hardware removes the least recently used block, guessing that it is unlikely to be referenced in the near future
  - The hardware must track references to each block
    - Use a counter for each block, cleared on each hit for that block, while others are incremented; replace the greatest-count block

Direct-Mapped Cache Example



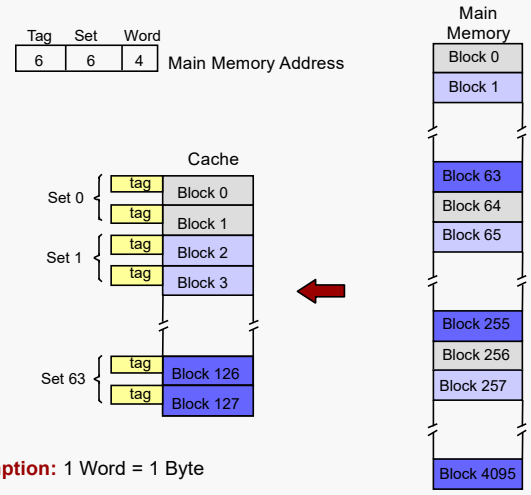
Assumption: 1 Word = 1 Byte

Associative Cache Example



Assumption: 1 Word = 1 Byte

Set-Associative Cache Example



Assumption: 1 Word = 1 Byte

Array Caching Example

Assumption: 1 Block = 1 Word = 1 Byte

Note: Typically, 1 Word = 4 Bytes, i.e., the least 2 significant bits of the memory address are used to identify a byte (simply ignore them during cache mapping).

Array Processing Program

```
SUM = 0;

for (j = 0; j <= 9; j++) {
    SUM = SUM + A[0][j];
}

AVE = SUM/10;

for (i = 9; i >= 0; i--) {
    A[0][i] = A[0][i]/AVE;
}
```


Direct-Mapped Cache State

Block position	Contents of data cache after pass:								
	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

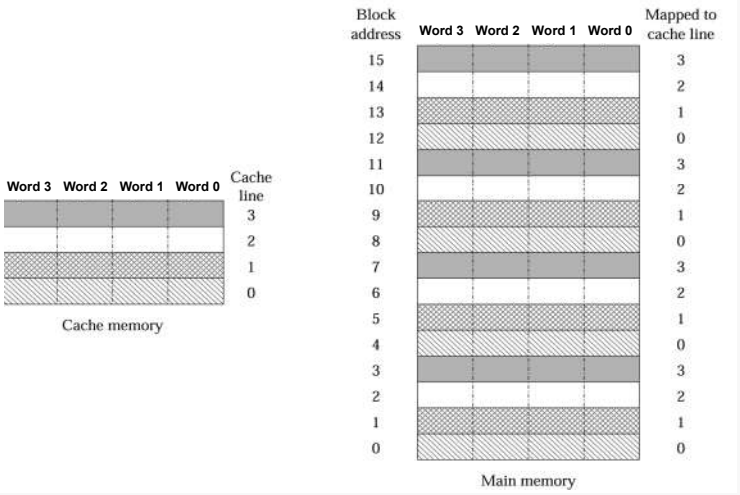
Associative Cache State

Block position	Contents of data cache after pass:				
	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

Set-Associative Cache State

	Contents of data cache after pass:					
	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Set 1						

Another Example I



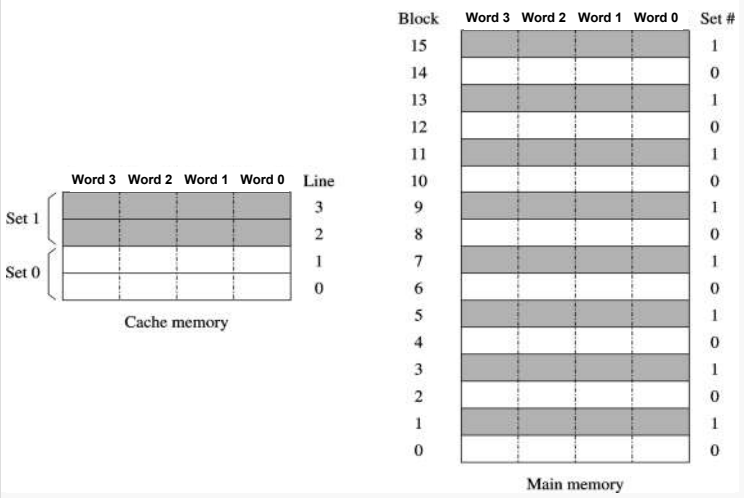
Another Example II

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???

Another Example III

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
7	Miss	Block 0	???	???	Block 7
9	Miss	Block 0	Block 9	???	Block 7
10	Miss	Block 0	Block 9	Block 10	Block 7
0	Hit	Block 0	Block 9	Block 10	Block 7
7	Hit	Block 0	Block 9	Block 10	Block 7
9	Hit	Block 0	Block 9	Block 10	Block 7
10	Hit	Block 0	Block 9	Block 10	Block 7
0	Hit	Block 0	Block 9	Block 10	Block 7
7	Hit	Block 0	Block 9	Block 10	Block 7
9	Hit	Block 0	Block 9	Block 10	Block 7
10	Hit	Block 0	Block 9	Block 10	Block 7

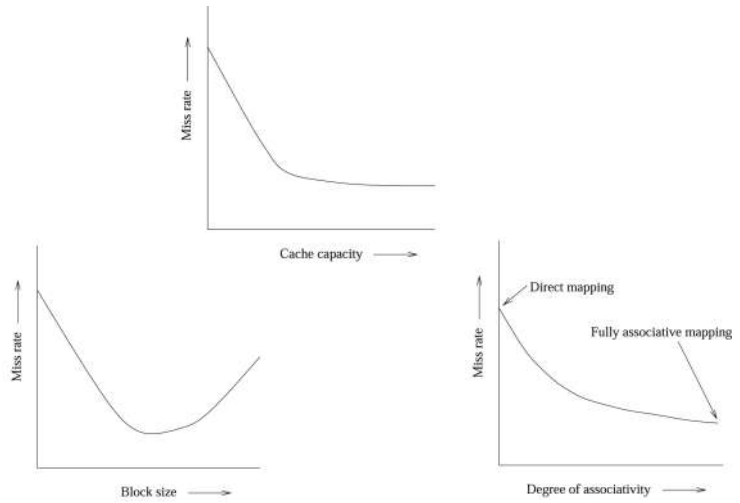
Yet Another Example I



Yet Another Example II

Block accessed	Hit or miss	Set 0		Set 1	
		Cache line 0	Cache line 1	Cache line 0	Cache line 1
0	Miss	Block 0	???	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
8	Miss	Block 0	Block 8	???	???
0	Hit	Block 0	Block 8	???	???
8	Hit	Block 0	Block 8	???	???
0	Hit	Block 0	Block 8	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
4	Hit	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
4	Hit	Block 0	Block 4	???	???

Cache Design and Cache Misses



Performance Enhancements

- **Write buffer**
  - When write-through is used, a write buffer can be included to temporarily store processor's write requests
    - The processor does not have to wait for a write to be completed to continue with an execution of the next instruction
    - Buffered writes are sent to the main memory when it is not responding to a read request (cannot stall reads!)
- **Prefetching**
  - Prefetch instructions are used to load data into the cache in advance, to avoid stalls on a read miss
- **Lockup-free cache**
  - Multiple outstanding misses can be supported
  - The processor can still access the cache, while a miss (in a different block) is still being serviced

Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves
  - **Temporal locality**: recently referenced items are likely to be referenced in the near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

```
sum = 0;
for (i = 0; i < n; i++) sum += a[i];
return sum;
```

- **Data**
  - Reference array elements in succession: **spatial locality**
  - Reference **sum** each iteration: **temporal locality**
- **Instructions**
  - Reference instructions in sequence: **spatial locality**
  - Cycle through loop repeatedly: **temporal locality**

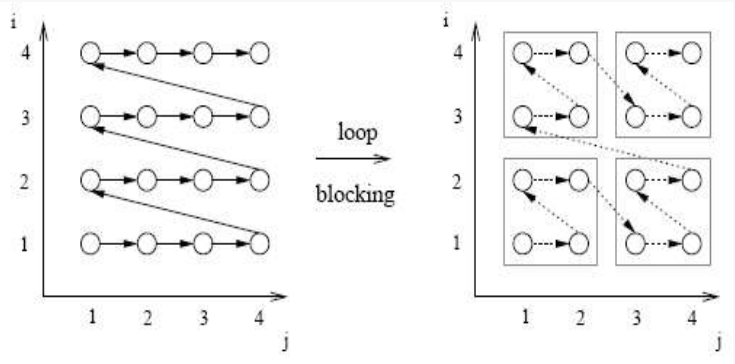
Cache-Efficient Coding

- Repeated references are good (temporal locality)
- Sequential references are good (spatial locality)
- Example:
  - $n \times n$ matrix transposition $\mathbf{a} = \mathbf{b}^T$ using B × B blocks

```
1: // Original code:
2: for i = 1 to n do
3:    for j = 1 to n do
4:       a[i,j] = b[j,i];
5:    end for
6: end for
```

```
1: // Loop blocked code:
2: for ii = 1 to n by B do
3:    for jj = 1 to n by B do
4:       for i = ii to min(ii + B − 1, n) do
5:          for j = jj to min(jj + B − 1, n) do
6:             a[i,j] = b[j,i];
7:          end for
8:       end for
9:    end for
10: end for
```


Blocked Transposition Example

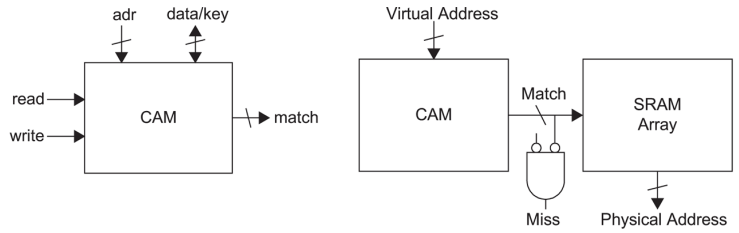


Original transposition: 4×4 matrix (n = 4)

Blocked transposition: 2×2 blocks (B = 2)

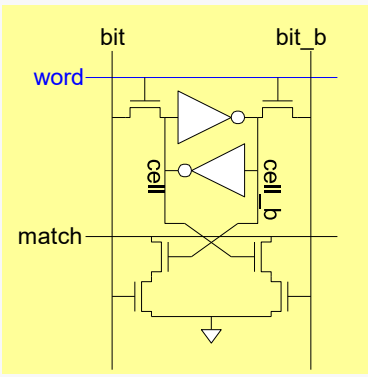
Content-Addressable Memory

- Extension of SRAM:
  - Read and write memory as usual
  - Also match to see which words contain a key



CAM Cell

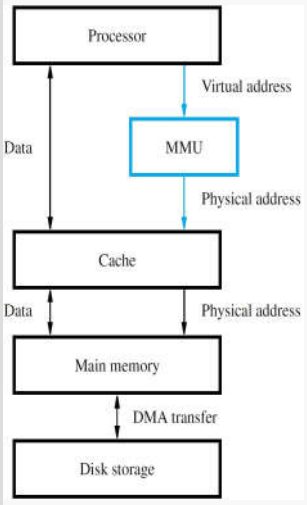
- Read and write same as ordinary SRAM
- Key matching:
  - Keep word line low
  - Precharge match line to 1
  - Place key on bit lines
  - Match line discharges to 0 if cell is different from key; otherwise, it remains at 1
- Example:
  - Let key = 1 → bit = 1 (bit_b = 0)
  - If cell = 0 (cell_b = 1), then match = 0; otherwise, match does not change (i.e., remains at 1)

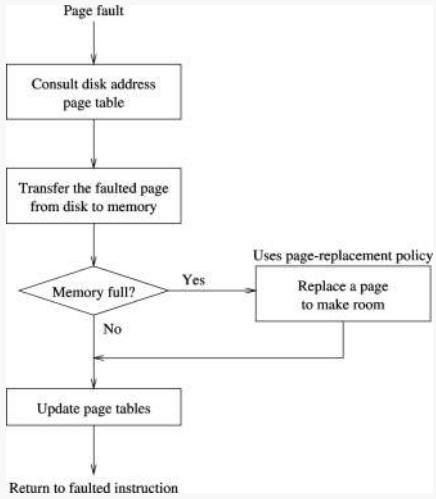


Virtual Memory I

- The physical main memory may be smaller than the full addressed space of the processor
- Secondary storage (e.g., disk) can be used to increase the apparent side of the physical memory
  - The processor generates a word's virtual address that is translated by the memory management unit (**MMU**) into the word's physical address
- If the addressed word is not in the main memory, it is said that a page fault has occurred
  - A multi-kilobyte page must be brought into the main memory (i.e., 'cached' from the secondary storage)
  - A modified page must be written back into the disk before it is removed from the main memory

Virtual Memory II

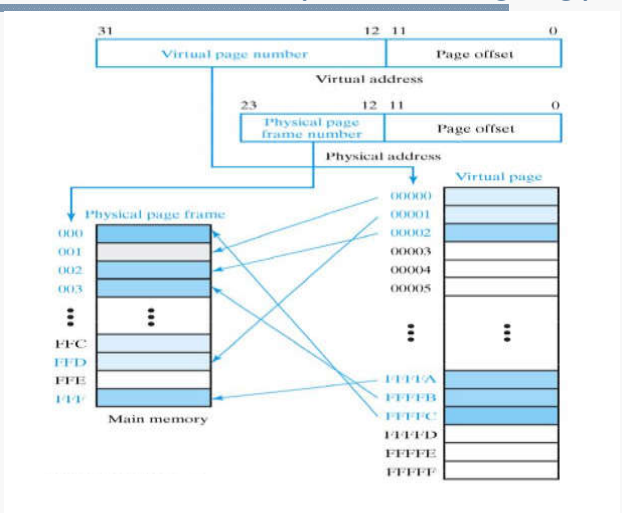




Page Faults

- MMU raises an interrupt for the OS to place the desired page (containing the requested word) into the main memory
  - The program that generated the page-faulting word's address becomes suspended
- Operating system selects an appropriate memory location for the incoming page
  - If the main memory is full, the OS will use a certain replacement policy (e.g., LRU) and, if needed, perform a write-back for the outgoing page
- **Note:** delay may be long, involving disk accesses, hence another *runnable* program is selected to run

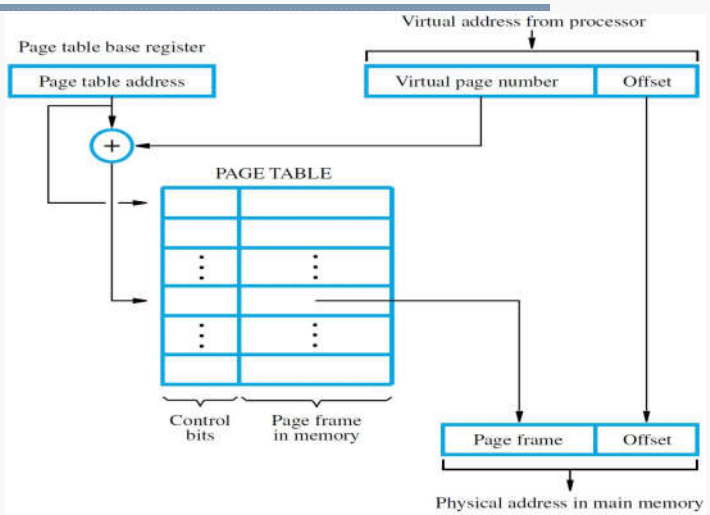
4 GB → 16 MB (4-KB Paging)



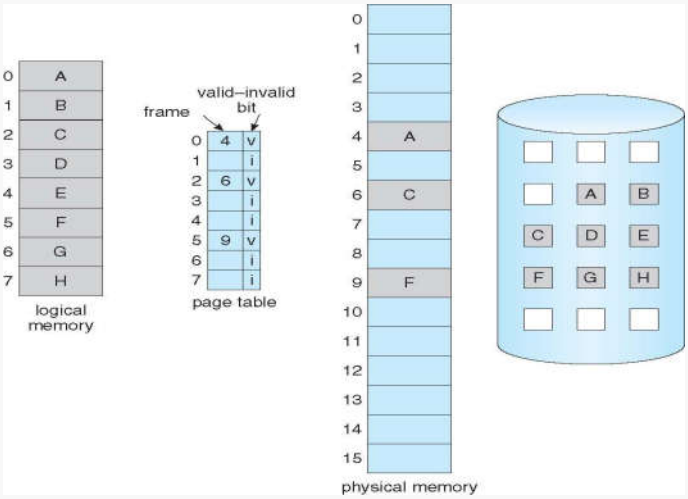
Virtual Address Translation I

- Virtual address of a word has 2 fields: **VPN** (virtual page number) + word's **offset** within a page frame
  - Translation preserves **offset** bits, but replaces **VPN** bits with page frame bits (i.e., physical page frame address)
- Page table (stored in the main memory) provides information needed for address translation
  - Page table base register has the starting address, and adding **VPN** to it finds the corresponding page entry
  - If the page is in memory, the page entry gives the page frame bits; otherwise, it may indicate disk location
  - Control bits for each entry include 'valid' (1 = resident in memory), 'modified' (1 = copy-back needed), and other bits (e.g., read/write permissions) associated with the corresponding page
    - **Note:** Do not confuse with 'valid' (0 = stale copy) and 'modified' (1 = stale original) bits associated with cache blocks

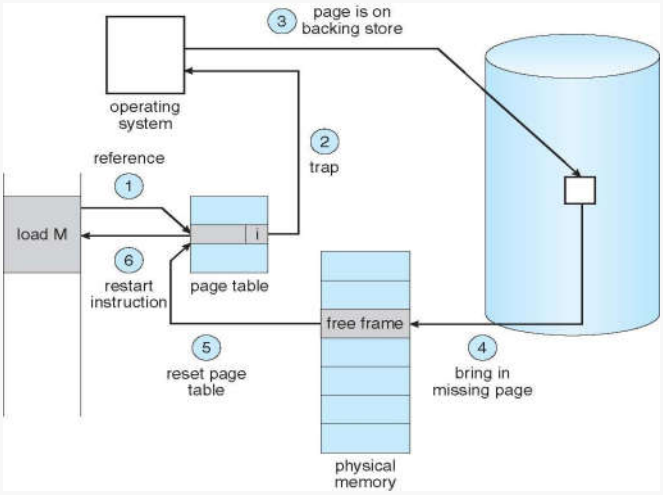
Virtual Address Translation II



Valid/Invalid ('valid' = 1/0) Bit



Page Fault Revisited



Page Sizing and Fault Rate

- Larger page size → more efficient disk access; smaller page size → less fragmentation, but…
  - Smaller page size → more pages required per process, more pages per process → larger page tables
  - Example: 4 GB → 16 MB (4-KB Paging)
    - Number of pages = 1 M (number of page table entries)
    - Page table entry size = 4 bytes → 4-MB page table!
- Average access time: $T_{ave} = (1 - p)M + pD$
  - Example:
    - Memory access time M = 200 ns
    - Page fault service time D = 8 ms (OS + disk access)
    - Page fault rate p = 0 → T_{ave} = 200 ns
      Page fault rate p = 1/400,000 → T_{ave} = 220 ns (10% slower)
      Page fault rate p = 1/1000 (0.1%) → T_{ave} = 8.2 µs (40x slower!)

Writing Good Code

- Example: sum elements of `int a[M][N]`
  - Array elements are stored (on disk) in row-major order
  - M = N = 1024 → each row (4N bytes) = one 4-KB page
  - Program has been allocated only 2 page frames (8 KB) in memory (one frame is for code and other data)
- **Good:** bring a row from disk, access N elements of that row → M (1024) page faults (i.e., p ≈ 0.1%)

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        sum += a[i][j];
```

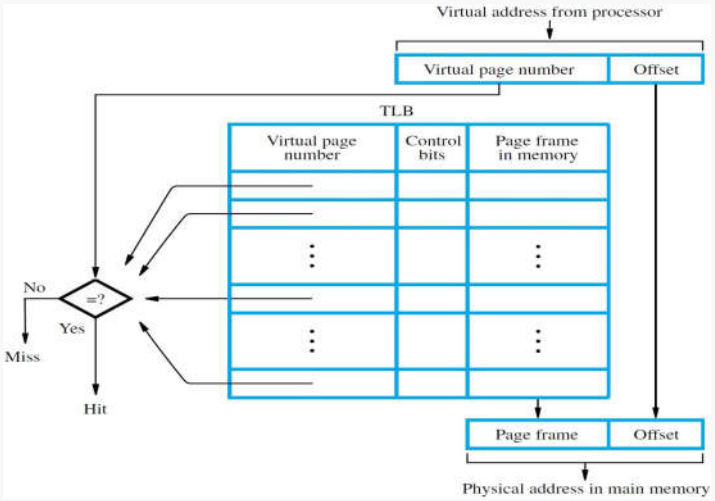
- **Bad:** bring a row from disk, access 1 element of that row → M×N (1,048,576) page faults!

```
for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
        sum += a[i][j];
```

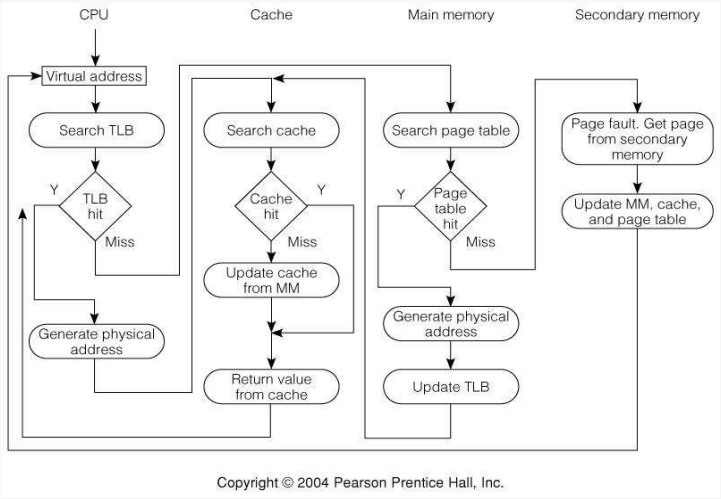
Translation Lookaside Buffer I

- MMU must know the page table's location in the main memory
- MMU must perform a lookup in the page table for translation of every virtual address
- For large physical memory, MMU cannot hold entire page table with all of its information
- MMU's translation lookaside buffer (**TLB**) holds recently-accessed entries of the page table
  - TLB = small fully-associative cache for the page table
- To find a matching entry for a given virtual address (tag), MMU performs a TLB search first
  - If a miss, access the full page table and update TLB

Translation Lookaside Buffer II



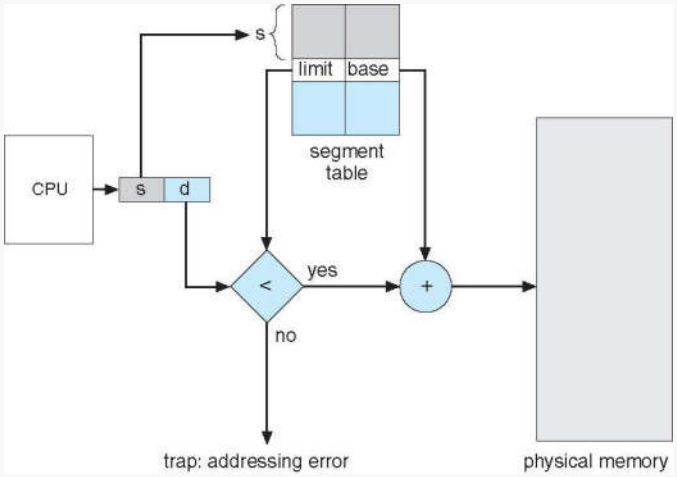
Putting It All Together



Segmentation I

- Program = collection of segments, e.g., stack, *main*(), user subroutines, library functions, etc
- CPU address: **s** (segment number) + **d** (offset)
- For each **s**, there is an entry in the segment table that specifies:
  - base – segment's physical address in the main memory
  - limit – segment length
  - read/write/execute privileges
- Segment-table base register points to the segment table's location in the main memory

Segmentation II



Example: Pentium