Locality of Reference

- Most programs exhibit a property known as “locality”
  - Addresses referenced tend to cluster

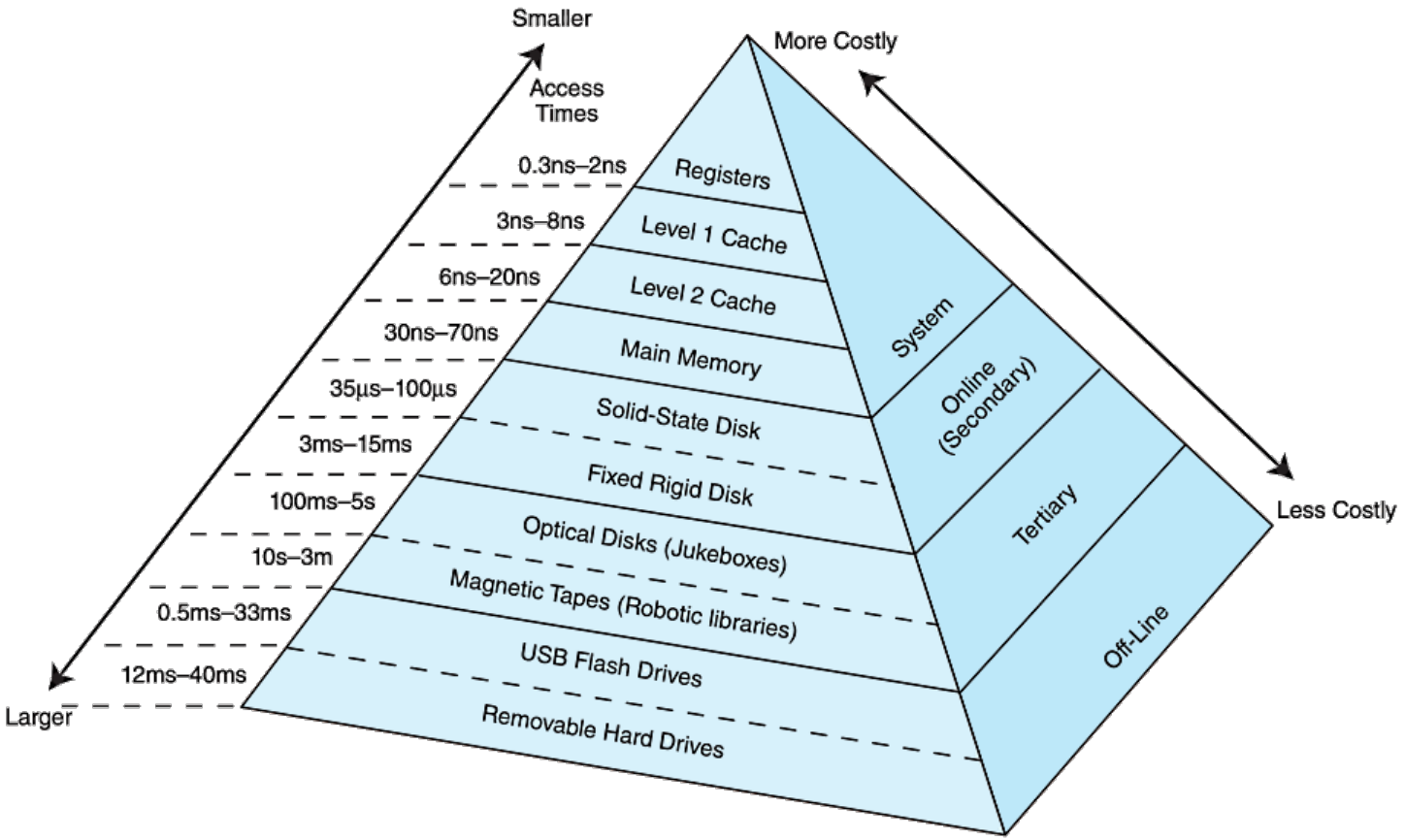
- Addresses within a relatively small area are reused
  - repetitive loops
  - fairly small functions
  - access to items within arrays and structures

- *Spatial locality*
  - the use of limited areas within the address space
  - these areas change or migrate slowly over time

Locality of Reference

- *Sequential locality*
  - Instructions tend to be accessed sequentially
  - The PC is incremented to point to the next instruction
  - Branches and function calls alter this sequential flow
- *Temporal locality*
  - Reuse of items recently accessed
  - Previously executed instructions
  - Previously accessed data operands
- Locality can be exploited to improve performance

Exploiting Locality



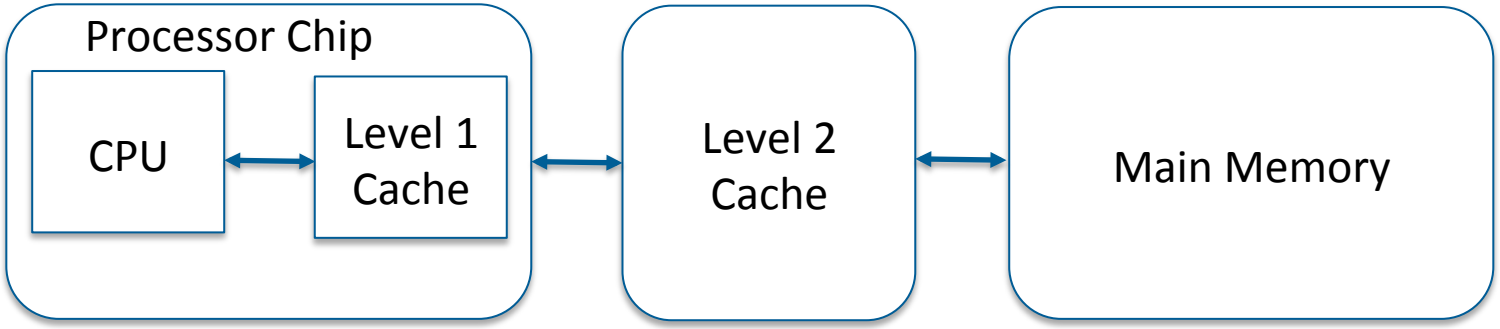
Keep recently used items at the higher levels within the memory hierarchy

Exploiting Locality

- Use registers for operands
  - RISC systems tend to have more registers
  - Registers are within the CPU
- Use a high speed “cache” memory
  - Constructed from SRAM
  - Much faster than DRAM (central memory)
  - Holds instructions and data
- Use multiple cache levels
  - Accomodates more instructions and data
  - Reduces frequency of access to central memory

Cache Memory

- L1 or primary cache is on the same chip as the CPU
  - Takes longer to access than registers
  - Much shorter access time than main memory
- L2 or secondary cache is not on same chip as CPU
  - Takes longer to access than L1 cache
  - Much larger size than L1 cache



- L1 is checked first, then L2, then main memory

Cache Memory

- Operands not in registers should be kept in cache
- Block containing a desired item is loaded into cache
  - This takes advantage of spatial locality
  - Blocks may contain hundreds of bytes
- Blocks are copied form main memory into L2 cache
- L1 contains a subset of the information in L2
- Main or central memory is accessed as a last resort
  - The extra time needed to access memory is a penalty

Cache Memory

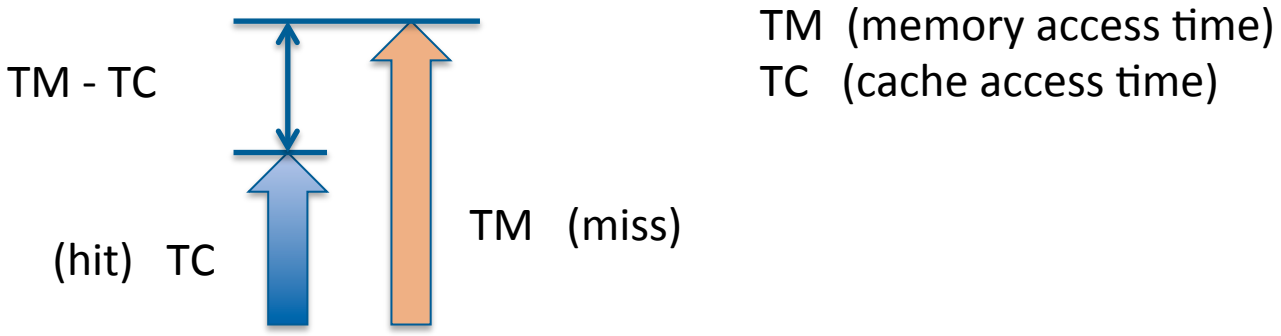
- Cache related terms:
  - A *hit* is when data is found at a given memory level
  - A *miss* is when it is not found
  - The *hit rate* = % of references found at a given memory level
    - Hit ratio = (Number of hits) / (total number of references)
  - The *miss rate* = % of references not found at that level
  - Miss ratio = 1 - hit ratio

Cache Memory

- Cache related terms:
  - *hit time* = time needed to access data at a given memory level
  - The *miss penalty* = time required to process a miss
    - includes block load and/or replacement time
    - includes time it takes to extract and deliver the item
  - The *effective access time* = average access time per reference
    - Depends on hit ratio, hit time and miss penality
- *Cache line* holds a block read in from main memory
  - Line size and block size are the same

Look-aside Caches

- These allow overlapped accesses
- Accesses to cache and to memory occur in parallel
- Memory access is cancelled if hit in cache occurs
- Tends to lower the effective access time
- Increases CPU-to-memory traffic



Look-aside Example

- Consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit ratio 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

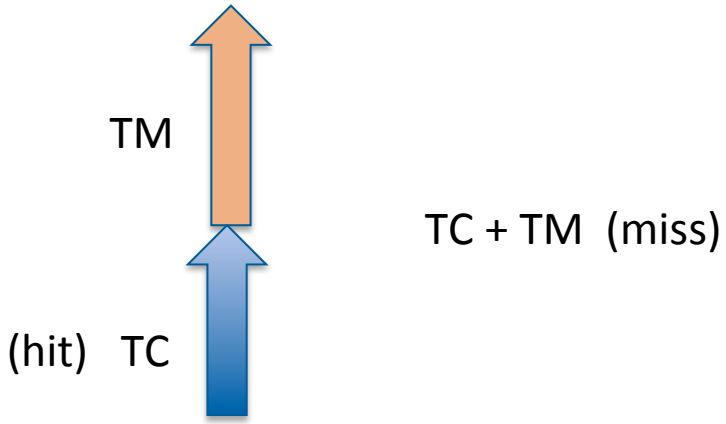
EAT = hit ratio*TC + (1-hit ratio)*TM

Look-through Caches

- With these, the cache is checked first
- Next level is only checked if miss occurs
- Tends to increase the effective access time
- Avoids unneeded CPU-to-memory traffic

EAT = h* TC + (1- h)*(TC+ TM)

where h is the hit ratio



Look-through 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:

$$0.99(10\text{ns}) + 0.01(10\text{ns} + 200\text{ns})$$
$$= 9.9\text{ns} + 2.01\text{ns} = 12\text{ns}$$

- This equation for determining the effective access time can be extended to any number of memory levels

Cache Issues

- Caching is beneficial if programs exhibit 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

Cache Issues

- Write Policy must be considered as well
  - *Two types: write through* and *write back (or copy back)*
- Write through updates cache and main memory
  - Memory is updated in parallel with cache for every write
- Write back only updates the cache copy of the item
  - Main memory is updated when item is removed from cache
  - Must record whether cache line was written (*dirty*)
  - May cause issues with concurrent access
    - Shared memory multiprocessors
    - I/O devices using DMA (direct memory access)

Cache Issues

- Cache contains only a fixed number of lines
- Lines are replaced to make room for new blocks
  - Required if all lines are already occupied
- Replacement Policy selects “victim” to be evicted
  - Dirty lines must be written back to memory
- Possible replacement policies
  - Random
  - FIFO (oldest)
  - LRU (least recently used)

Unified versus Split Caches

- *Unified* or *integrated* caches contain instructions & data
- The alternative is a *split cache* system
  - Instruction (I-cache) and data (D-cache) are separate
  - This is called a *Harvard* cache
  - Allows instruction fetch in parallel with data access
- 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.

Multilevel Cache

- Many modern systems use multiple levels of cache
- L2 is on the same die as the CPU in 3-level systems
- Level 3 cache is between the CPU and main memory
- Some systems allow multiple copies of information
  - data and instructions are in more than one cache level at a time
  - These are called inclusive cache systems
- Exclusive caches permit only one copy of data

Sources of Misses

- Compulsory Misses
  - Also called “cold start” misses
  - Unavoidable due to initially empty cache
- Capacity Misses
  - Due to limited size of cache
  - Cache too small to hold all referenced blocks
- Conflict Misses
  - Due to different blocks mapping to the same set or line

Conclusion

- Next we will consider cache mapping schemes
  - Determines the line into which to load a block
  - Depends on the cache organization
  - Affects the line replacement policy
- We will exam three organizations
  - Fully associative
  - Set associative
  - Direct mapped

Cache Organization

Cache makes memory appear to be faster

- It is closer to the CPU than is main memory
- It holds recently accessed instructions and data

- Cache is much smaller than main memory
- Its access time is a fraction of that of main memory
- Main memory is accessed by address
- Cache is accessed by content
  - it is often called *content addressable memory*
- A single large cache isn't always desirable–
  - it takes longer to search.

Cache Organization

- We will exam three cache organizations:
  - Fully associative
  - Set associative
  - Direct mapped

- Each is based on the idea of memory blocks
  - A program's address space is subdivided into blocks
  - Each block is like an element in an array
  - Each is assigned a unique block number (B)
  - The block address = B * block size

  (e.g. if block size = 256, block 0 begins at address 0, block1 begins at address 256, etc.)

Cache Organization

- Addresses of items map to blocks
- Address can be interpreted as:

  - Block number = address / block size
  - Number of bits in offset = $\log_2$ block size
  - Example: for 256-byte blocks, the address 0x0400ACE8 falls within block:
    0x0400ACE8 / 256 = 0x00400AC
    Offset width = $\log_2$ 256 = 8 bits

Cache Organization

- Cache can hold some number of blocks
  - The blocks within the cache are called lines
  - Each line is the same size as a memory block
  - A line can hold one of a collection of blocks
  - Mapping determines which block resides in which line
- Number of lines is much less than number of blocks
  - (Memory size) / (block size) = number of blocks
- Cache size determines the number of lines
  - Example: If cache size = 8 KB and line size = 256 bytes, then number of lines = 8 KB / 256 = 32

Cache Organization

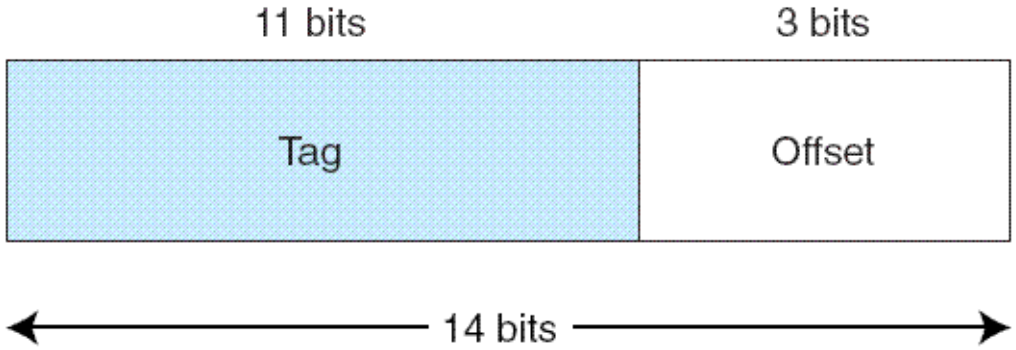
- A block is the basic unit of transfer between memory and cache
- A miss results in copying the entire block to cache
  - Takes advantage of spatial locality
- Cache is organized into some number of sets
- *Associativity* is the number of blocks per set
  - N-way associative means there are N blocks per set

Fully Associative

- Fully Associative is the easiest to describe
- Any memory block can go into any empty cache line
- All lines are checked to detect cache hits or misses
- Requires hardware to compare all lines in parallel
  - A valid bit for each line is set if the line is occupied
  - Block loaded into vacant cache line if miss occurs
  - If no line is vacant, a replacement must occur

Fully Associative

- Mapping is based on dividing address into 2 fields:
  - Tag field and offset



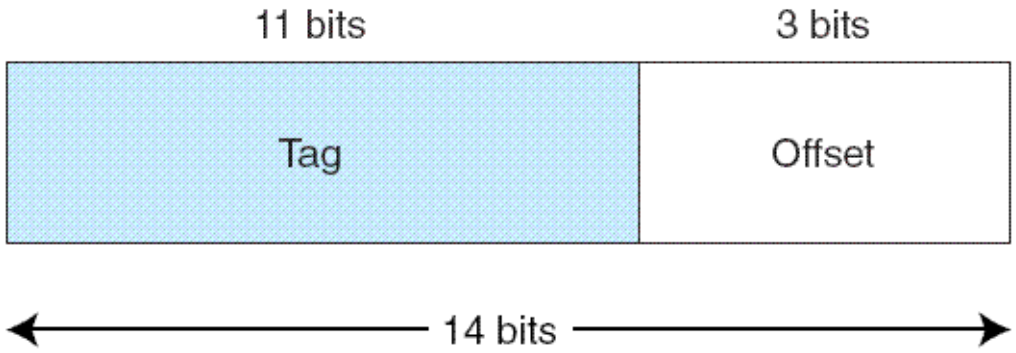
Example: 14-bit addresses and a cache containing 16 lines. Each line is 8 bytes in size.

A separate tag is stored for each cache line

If a stored tag matches the tag field in the address and the line is valid, there is a hit.

Fully Associative

A separate tag is stored for each cache line

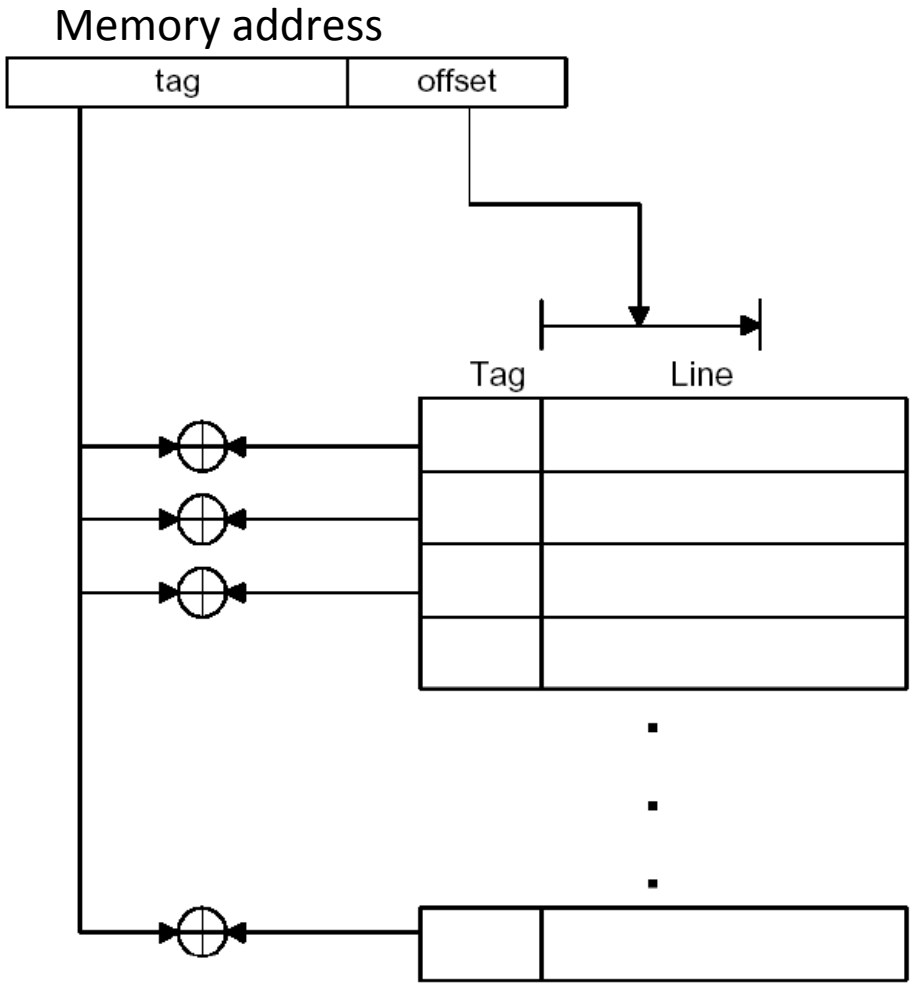


Only lines for which valid bit is set are checked

Match between a stored tag and address tag indicates a hit

Miss means none of the stored tags match the address tag

Fully Associative



Separate comparator is needed for each cache line

Example

- Assume a read and a block size of 256 bytes
- Address 0x0400ACE8 is interpreted as:

  - Block number = address / block size = tag
    0x0400ACE8 / 256 = 0x00400AC
  - a hit occurs if some cache line has a matching tag
  - for a miss, the block is loaded into any empty line
  - if all lines are full, a line is replaced

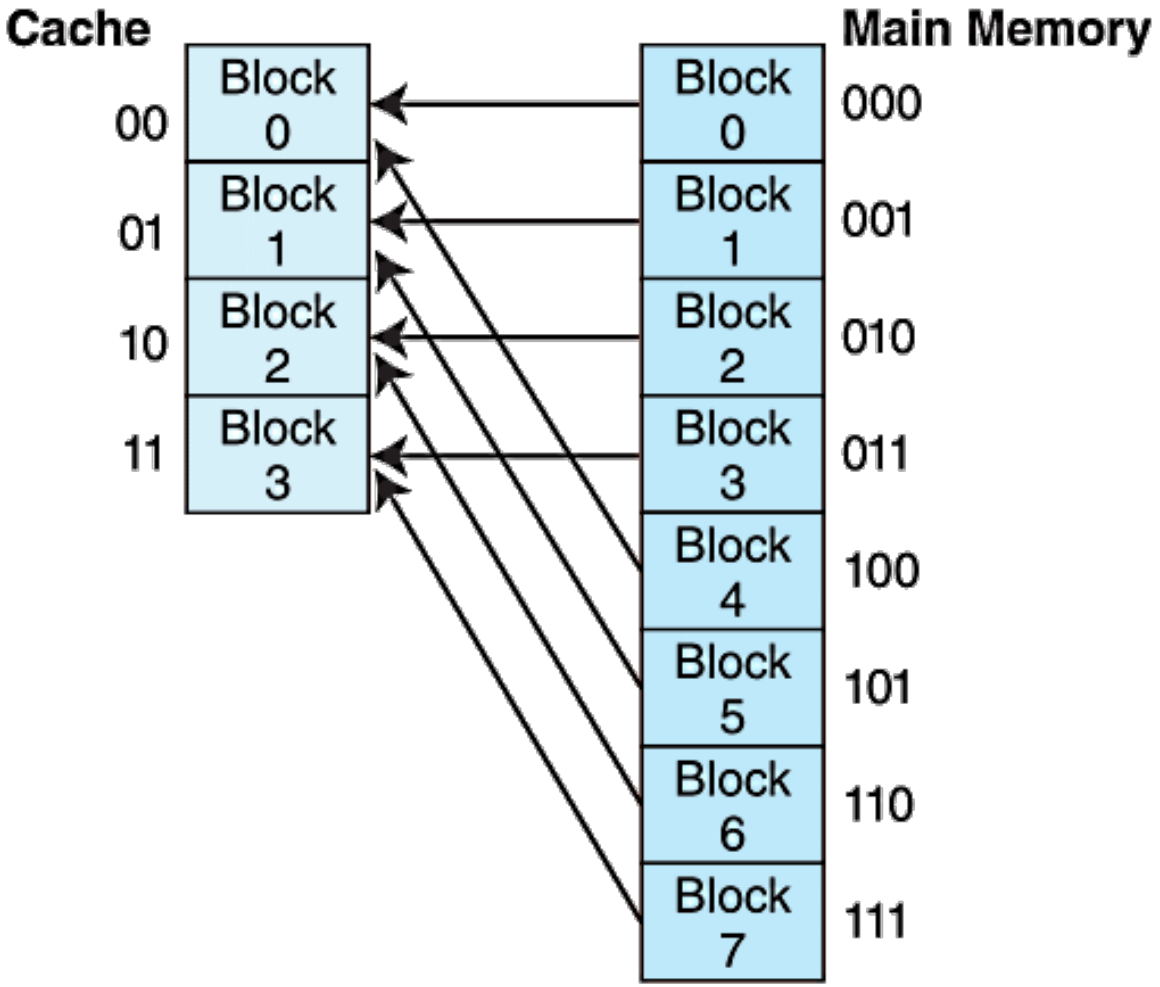
Direct Mapped

- Fully Associative is one extreme
  - A separate comparator is required for every cache line
- Direct Mapped is the other extreme
  - only one comparator is required for entire cache
- With a direct mapped cache with N lines:
  - block B of memory maps to line L = B mod N
  - B is the block number, not the address

Example: direct mapped cache with 10 lines,
Line 6 may hold blocks 6, 16, 26, 36, …

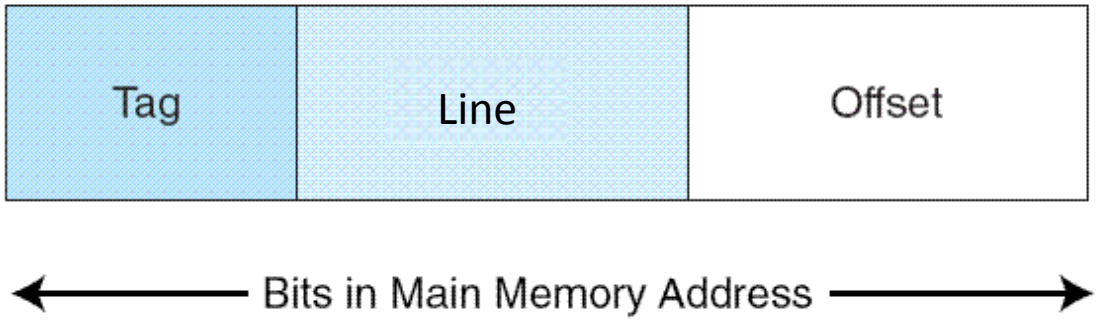
Direct Mapped

- With direct mapped cache consisting of *N* lines of cache, block *X* of main memory maps to cache line *Y* = *X* mod *N*.



Direct Mapped

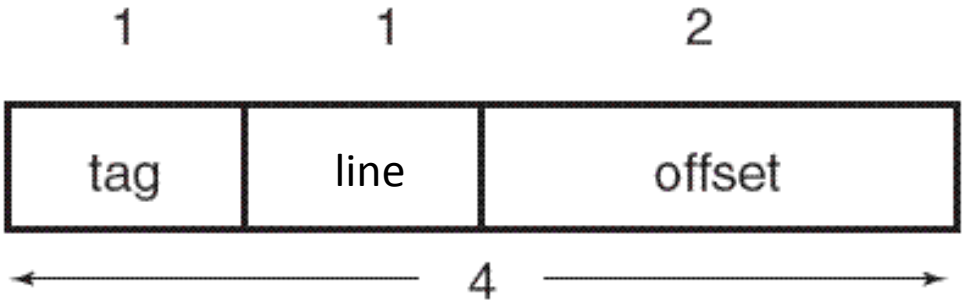
- Address is split into 3 fields for direct mapping
  - The *offset* field identifies location within line
  - The *line* field selects a unique line of cache
  - The *tag* field is whatever is left over.



- Offset width OW = $\log_2$ line size
- Line# width LW = $\log_2$ number of lines
- Tag width = (Bits in address) - LW - OW

Direct Mapped

- EXAMPLE: byte-addressable main memory consisting of 4 blocks with 4-byte block size. Cache contains 2 lines.
- Block 0 and 2 of main memory map to line 0 of cache, and Blocks 1 and 3 of main memory map to line 1 of cache.
- Address format is:



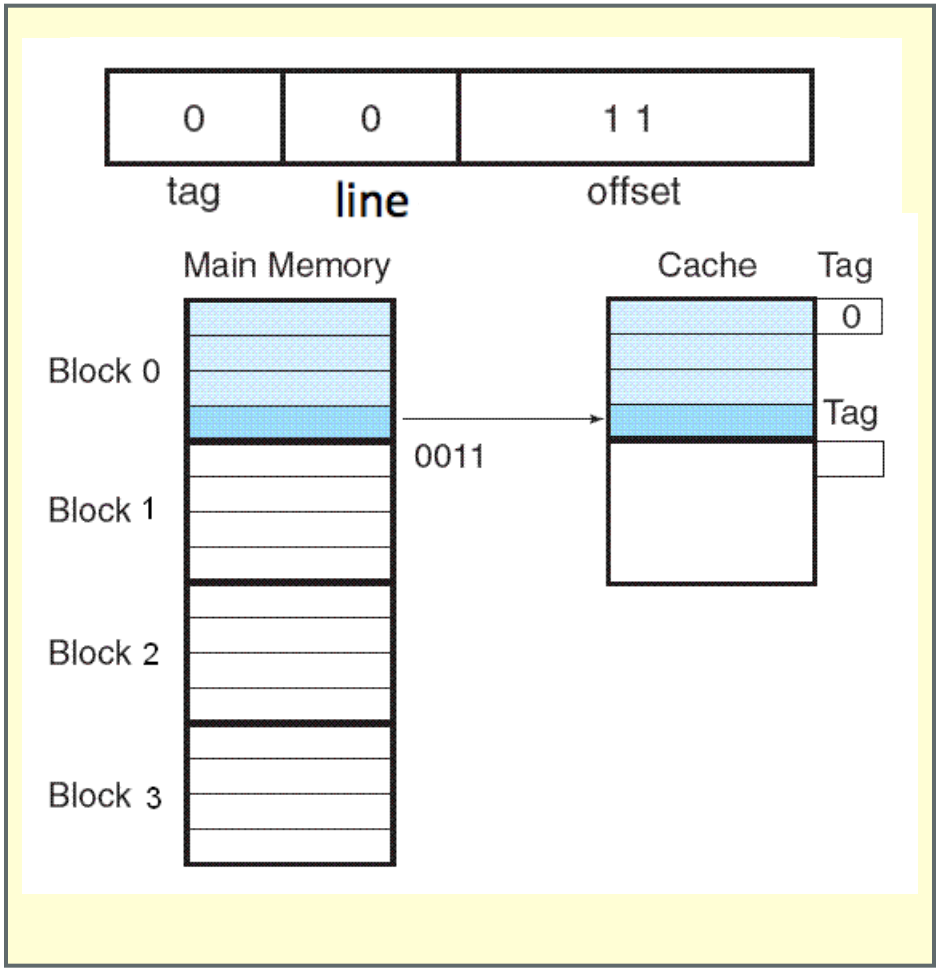
Direct Mapped

Address 3 is in memory block 0 and maps to cache line 0

Block# = 3 / 4 = 0

line# = 0 mod 2 = 0

16-byte memory
Requires 4-bit address



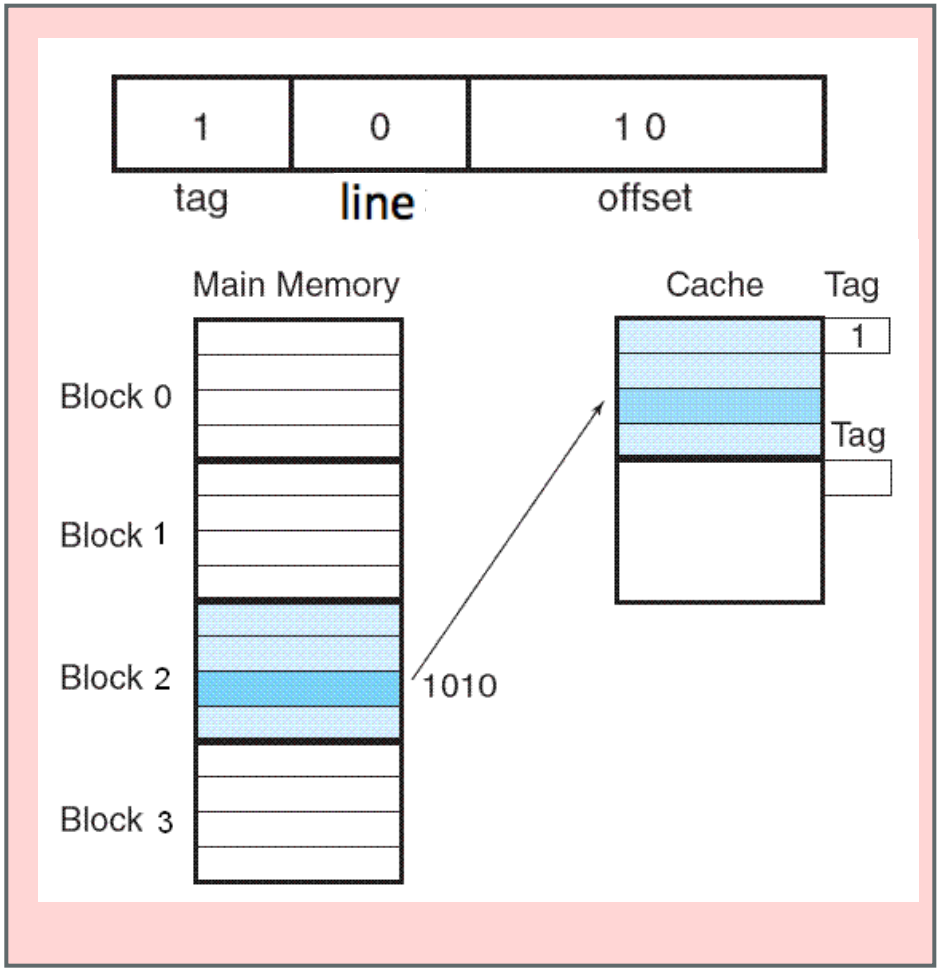
Cache has 2 lines:
line0 and line 1

Direct Mapped

Address 10 is in memory block 2 and maps to cache line 0

Block# = 10 / 4 = 2

line# = 2 mod 2 = 0



Direct Mapped

- Multiple addresses with same line number cause conflicts
  - cache line can hold only one candidate block
  - other vacant lines may be unused
  - causes increased miss ratio

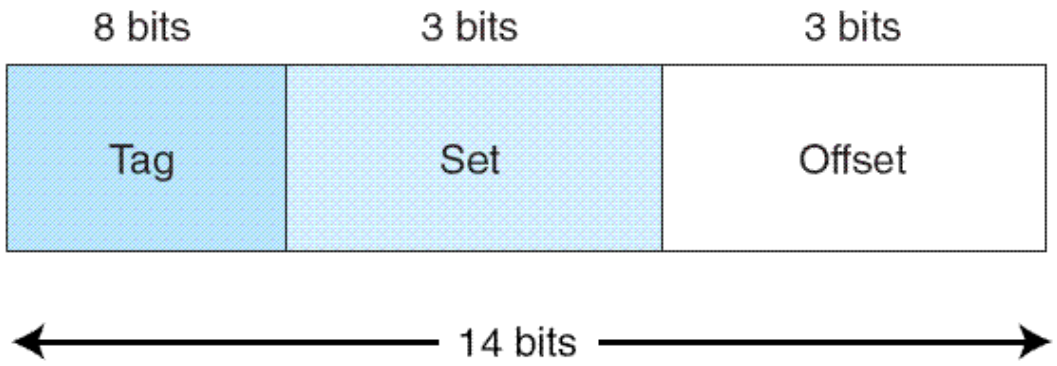
- Can hurt performance
  - 0 hit ratio if alternating between conflicting addresses

Set Associative Cache

- Set associative cache mapping reduces conflicts
- Sets of blocks map to the same set of lines
- Addresses are divided into three fields:
  - tag, set, and offset
- set field determines set (group of lines) to which the memory block maps
- tag field identifies which line within set contains block
- offset field chooses the location within the line

Set Associative Cache

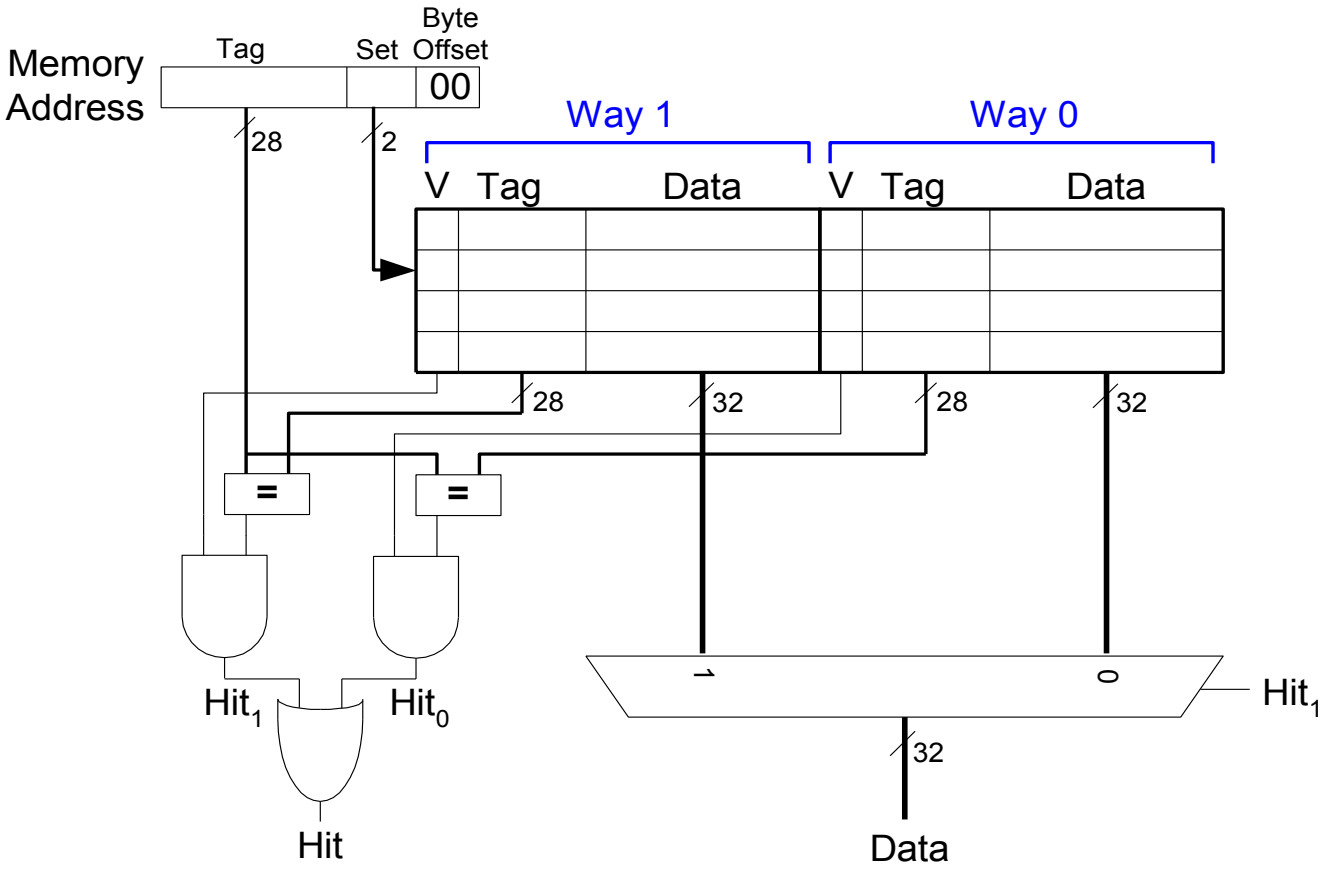
- EXAMPLE:
- 2-way set associative cache (2 lines per set)
- 2^{14} bytes of main memory (14-bit addresses)
- Cache with 16 lines, 8 bytes per line, sets = 16/2 = 8



- Address 0x12CF = 01001011001111 as 14-bit binary
- Maps to set 1
- Hit if tag for either line in set 1 = 01001011 = 0x4B
- Line must be valid (i.e. contain valid data)

Set Associative Cache

Example: 32-bit address, 4 sets, 2 lines per set, 4 bytes per line



Set Associative Cache

- New block can be loaded into any vacant line in set
- Replacement is required if set is full
- Most common choice is LRU (least recently used)
- O*ptimal* replacement policy
  - replaces line not needed in the future
- Optimal policy can't be implemented
- Used as benchmark for assessing other schemes
- Replacement policy requires extra bits for each set

Replacement Policies

- Replacement policies try to optimize temporal locality
- *First-in, first-out* (FIFO) is a popular replacement policy
  - oldest line in the set is evicted
  - ignores when it was last used
- *Random* replacement policy
  - Replaces line at random from the set
  - Can evict a line that will be needed often or soon
  - it never *thrashes*
- Trashing refers to repeatedly replacing the wrong line

Replacement Policies

- *Least recently used* (LRU) is most popular
- Keeps track of the last time a line was accessed
- Line unused for the longest period of time is evicted
- Disadvantage is its complexity
- Maintaining access history for each line, slows down the cache
- Can be approximated to save bits

Pseudo LRU Replacement

- *True LRU replacement*
  - Always correctly identifies the actual least recently used line
  - Requires a number of bits per set (depends on associativity)
  - Can be excessive in terms of overhead (storage & speed)
- *Pseudo LRU* replacement
  - Identifies least recently used line most of the time
  - Can make wrong choice at times
  - requires fewer bits per set
  - Faster than true LRU

Pseudo LRU Replacement

Example: Scheme used on the Pentium and other processors with 4-way associative cache
3 bits (B2 B1 B0) are stored for each set
Bits are updated based on way (line) accessed:

Way accessed	Effect on LRU bits
0	1->B1, 1->B0
1	0->B1, 1->B0
2	1->B2, 0->B0
3	0->B1, 0->B0

Reading, writing or replacing a line is considered an "access"

LRU bits are updated for each access

New block is loaded into next available line
Replacement occurs if the set is full

Pseudo LRU Replacement

LRU bits select the victim when a replacement is required:

B2 B1 B0	Way to replace
000	0
001	2
010	1
011	2
100	0
101	3
110	1
111	3

Correct selection >90% of the time
Occasionally does not select the actual LRU way

Summary

- Set associative is the most general case
  - Number of lines per set = associativity
- Direct mapped is equivalent to a 1-way set associative cache
  - Each line is a separate single-element set
- Fully associative has 1 set with N elements
  - N = number of lines in the cache

Virtual Memory

- Cache is used to make memory appear to be faster
- Virtual memory makes main memory appear larger
- Virtual memory encompasses the secondary storage
  - The currently used portion of the program is in memory
  - The remaining parts stay on disk until needed

Virtual Memory

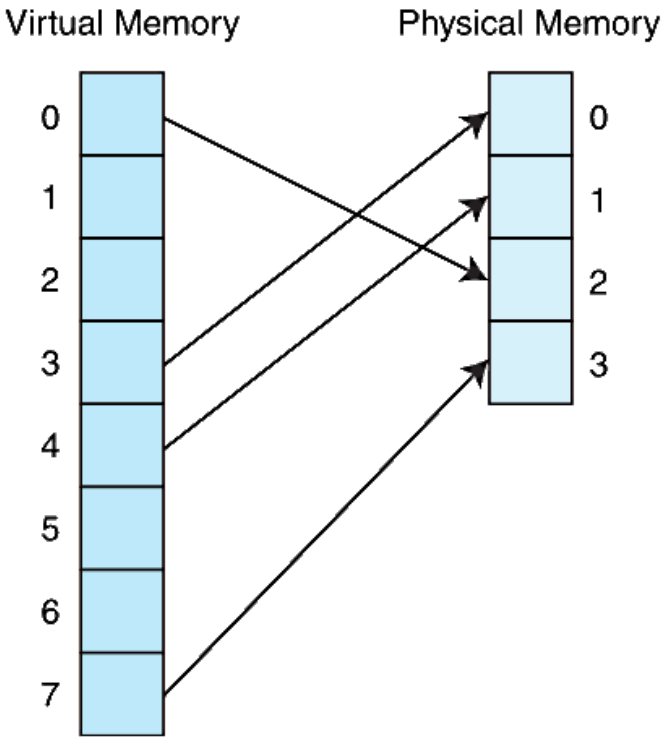
- Most virtual memory systems employ paging
  - Programs use a logical address space
  - Logical address space is partitioned into pages
  - Physical memory is partitioned into frames (same size as page)
- Main memory and virtual memory are divided into equal sized pages
- Pages are allocated to a process or program
  - Pages do not need to be stored contiguously-- either on disk or in memory

Virtual Memory

- Addresses used by the program map to pages
  - Page Map Tables are used to do the mapping
  - References to pages not in memory cause “page faults”
- There are many more pages than frames
  - This gives the illusion of a larger memory

Virtual Memory

There is one page table for each active process



Page Table (PMT)

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

PTEs

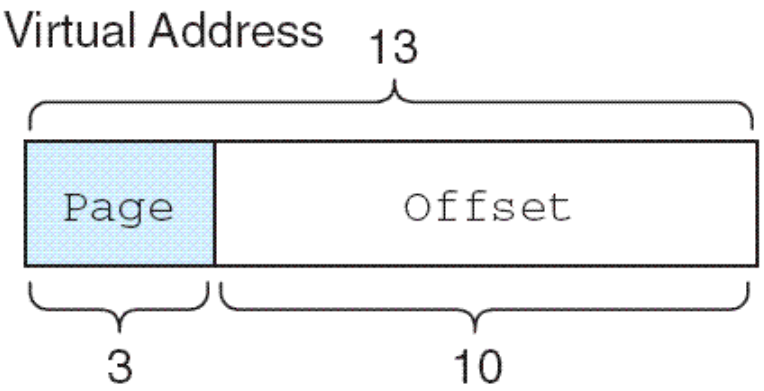
If valid bit=1, the page is in memory, and the page table entry (PTE) indicates which frame contains the page

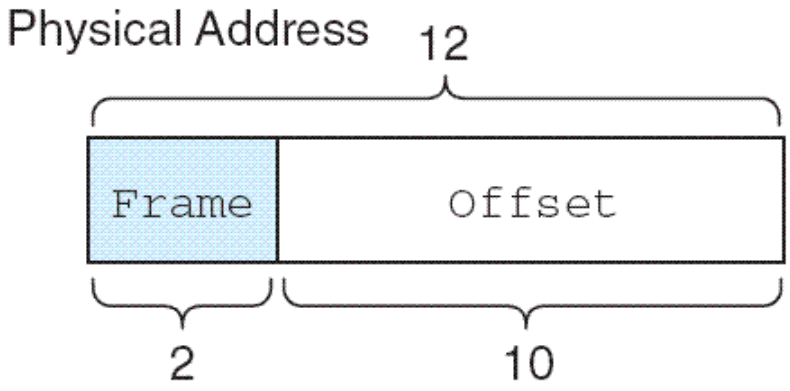
Virtual Memory

- Operating systems translate virtual addresses
- A virtual address is divided into two fields:
  - *page* number field, and
  - *offset* field
- page number indexes into the PMT to select PTE
  - If valid bit = 0 within PTE, we have a page fault
  - If valid bit = 1, the frame number in PTE is read
    - Frame number concatenated with offset = physical address

Virtual Address Translation

- Example:
- a byte addressable system has an 8K virtual address space, a 4K physical address space with 1K frames
- We have $2^{13}/2^{10} = 2^3 = 8$ virtual pages
- A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 bits for the page offset
- A 4K physical memory address requires 12 bits, the first two bits for the frame and the trailing 10 bits for the offset





Virtual Address Translation

- Assuming the PMT contents shown below, what happens for CPU address $5459_{10} = 1010101010011_2$ = 0x1553?

Page = 5

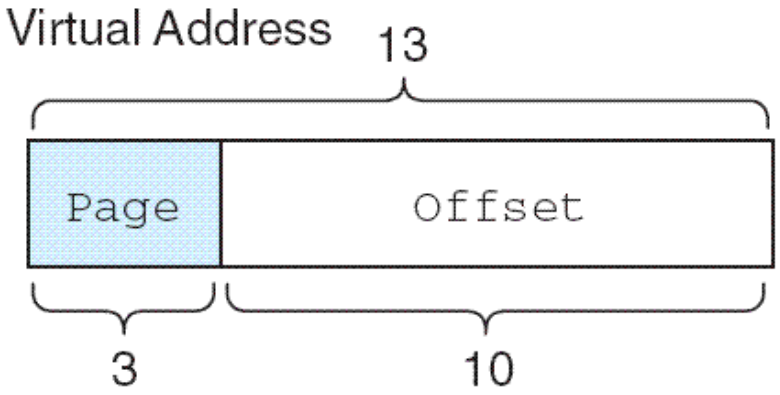
Page Table

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

Virtual Address Translation



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

Virtual Address Translation

The address 1010101010011_2 is converted to physical address 0101010010011_2 = 0x553 because the page field 101 is replaced by frame number 01 through a lookup in the page table (PTE 5)

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

Virtual Address Translation

- Note that the use of the PMT causes 2 accesses
  - The first access is to the PTE to get the frame
  - The resulting physical address is then accessed
  - Ex: lw $8,40($5) now accesses memory twice to load the operand
- Suppose main memory access time = 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk. Effective access time is:

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

Even with 0% page faults, EAT = 400ns

Virtual Address Translation

- Cache can be used to speed up the translation
  - Recent translations can be saved in cache
  - This type of cache is a translation look-aside buffer (TLB)
  - Can be implemented as an associative cache
  - Its hit ratio is high due to the locality property
- All TLB entries are checked in parallel for the page
- A TLB hit provides the frame number immediately
- A TLB miss means that the PMT must be checked

TLB Lookup Process

1. Extract page# and offset from the virtual address.

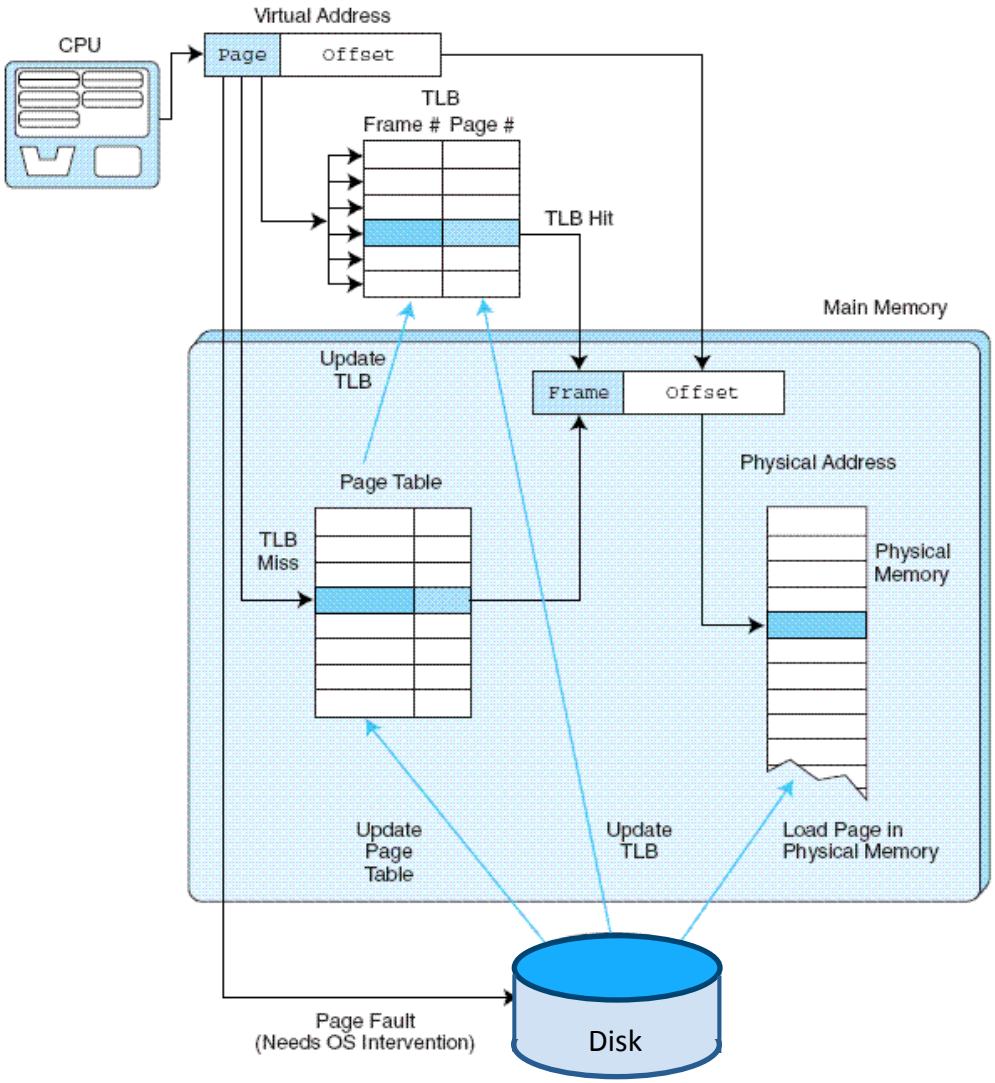
2. If TLB hit, combine frame# from TLB with offset to yield the physical address

3. Else use page# to check the page table entry (PTE) in the page map table (PMT).

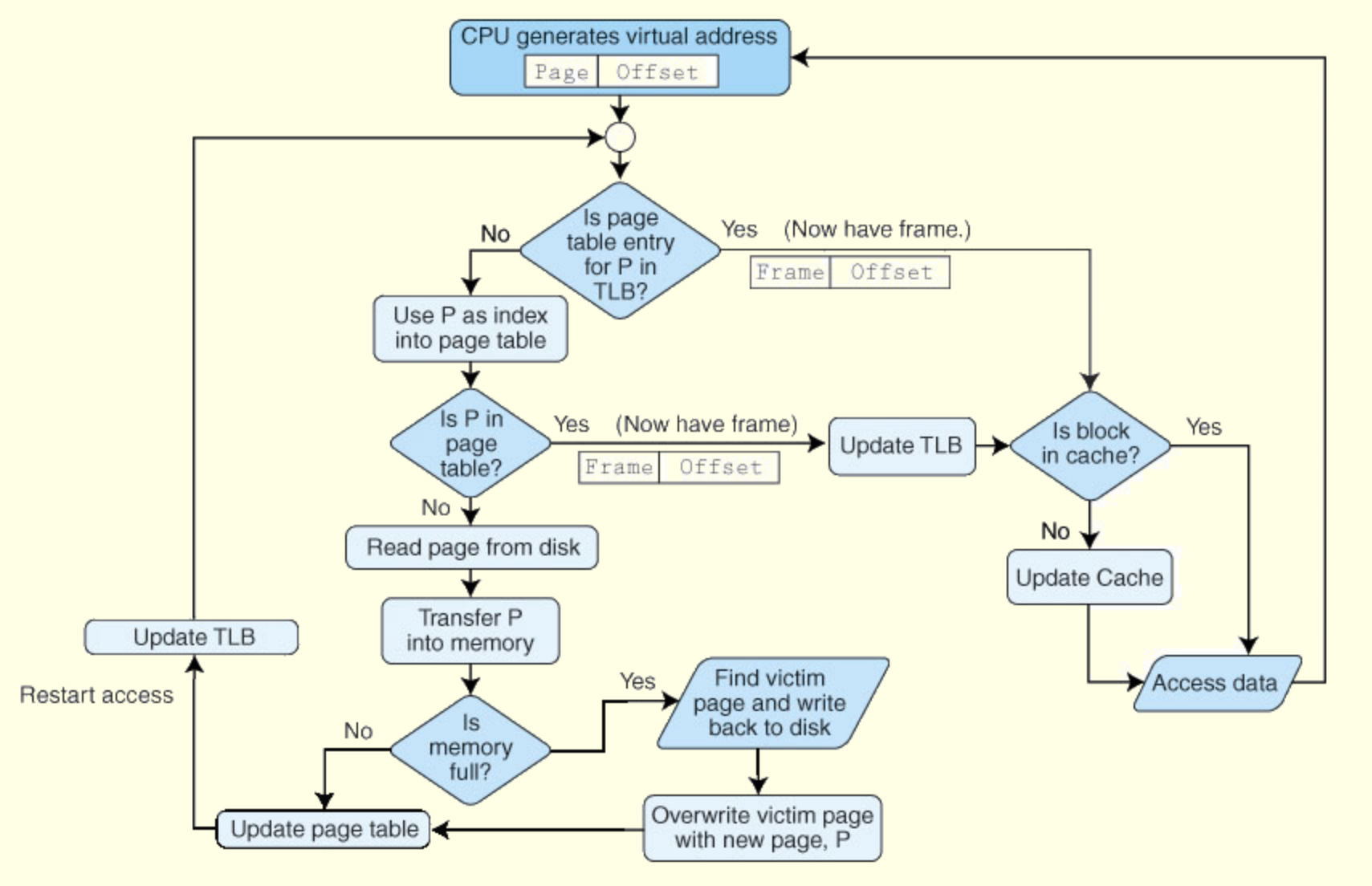
If valid bit is set, combine frame# with offset to yield the physical address.

4. Else generate a page fault and load page from disk.

5. Restart the access once page is in memory.



Virtual Memory In Action



Virtual Memory

The Operating System must identify free frames
- Bit string with one bit per frame (0=free, 1=occupied)
- Linked list with one node per frame

Replacement occurs if all frames are full
- Algorithm can be implemented in software
  - LRU (access bit periodically cleared by OS)
  - Random
- Only modified pages are written back to disk

Virtual Memory

Page faults can result from writes as well

Pages are loaded then updated for write misses

Write-through is never used due to slow disk access

Modify bit (dirty bit) is set for each write to a page

A bit within the PTE is also set whenever a page is accessed

- The access bit is cleared periodically by the OS
- Access bit = 0 if page has not been recently used

Memory Protection

The OS insures that frames are assigned exclusively

- Each process is assigned a different set of frames
- PTEs contain ID indicating owner of the page

Processes can reference same virtual address without conflict

Different tasks can request to share pages

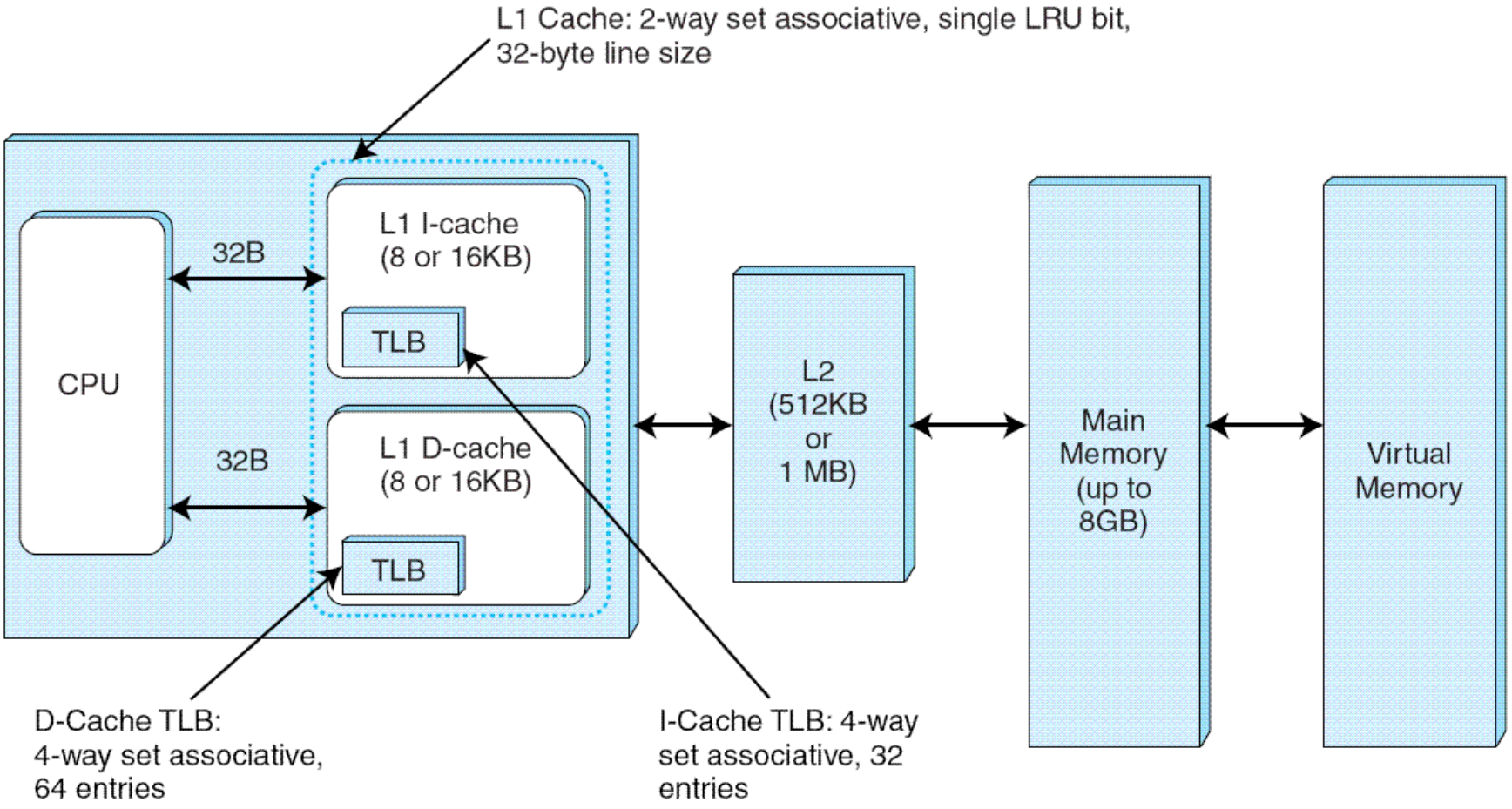
- Requires OS assistance

Hardware supports protection

- Privileged supervisor mode
- Privileged instructions
- PMT is only accessible in supervisor mode

Example System

Pentium System with multiple caches and virtual memory



Multi-Level Page Map Tables

- PMTs contain an entry for each possible page
- Address space may correspond to millions of pages
- Some programs only use a subset of the pages
  - Only pages actually used need to be in memory

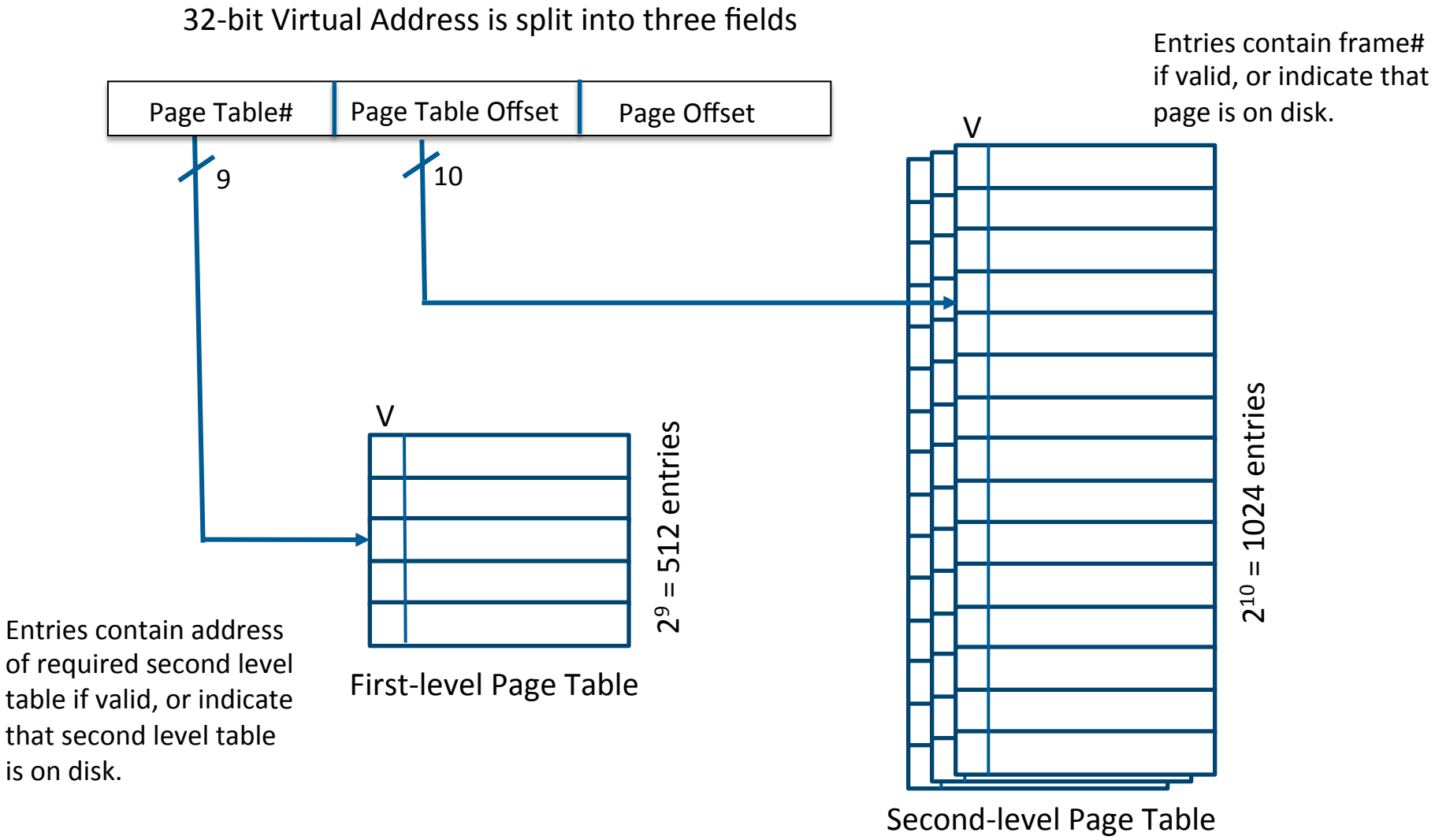
Multi-Level Page Map Tables

- Multi-level Page Tables can reduce the required space
  - Only the portion of the PMT that map the pages used
  - The unused portions can reside on the disk
  - The PMT itself can be paged
- By paging the PMT, less memory is used
- This works by partitioning the page number field

Virtual Memory

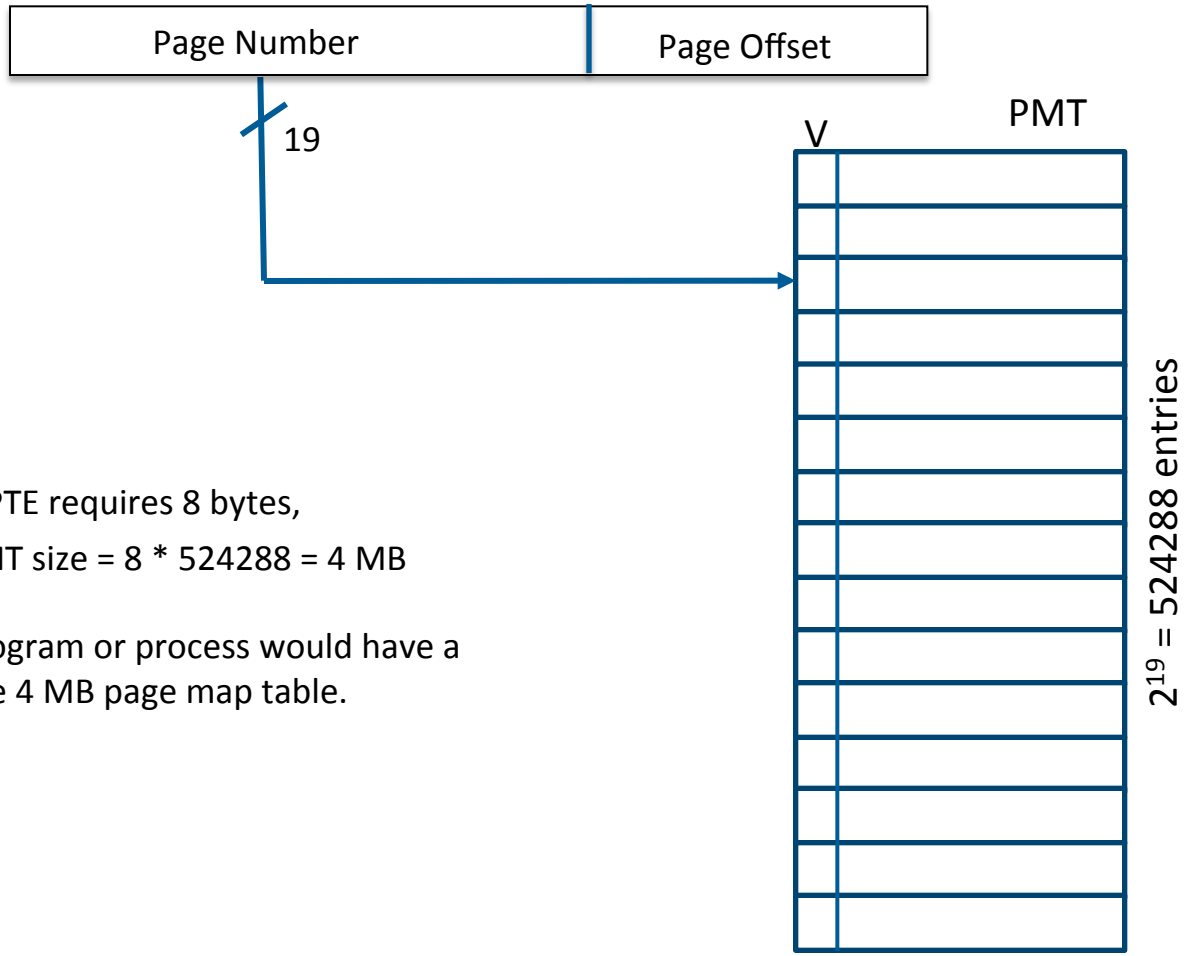
- A two-level system (page table & page offset):
  - Page table field indicates entry in first level table
  - Page table offset indicates entry in second level table
  - Page offset indicates location within selected page
- Three levels could also be used
- The first level table is kept in memory
  - Lower level tables are read in on demand
  - The final level contains the actual frame numbers

Example 2-level System



Example 2-level System

Using equivalent single PMT would require 2^{19} table entries



Entire PMT is in memory, even if not all entries are used.

If each PTE requires 8 bytes,

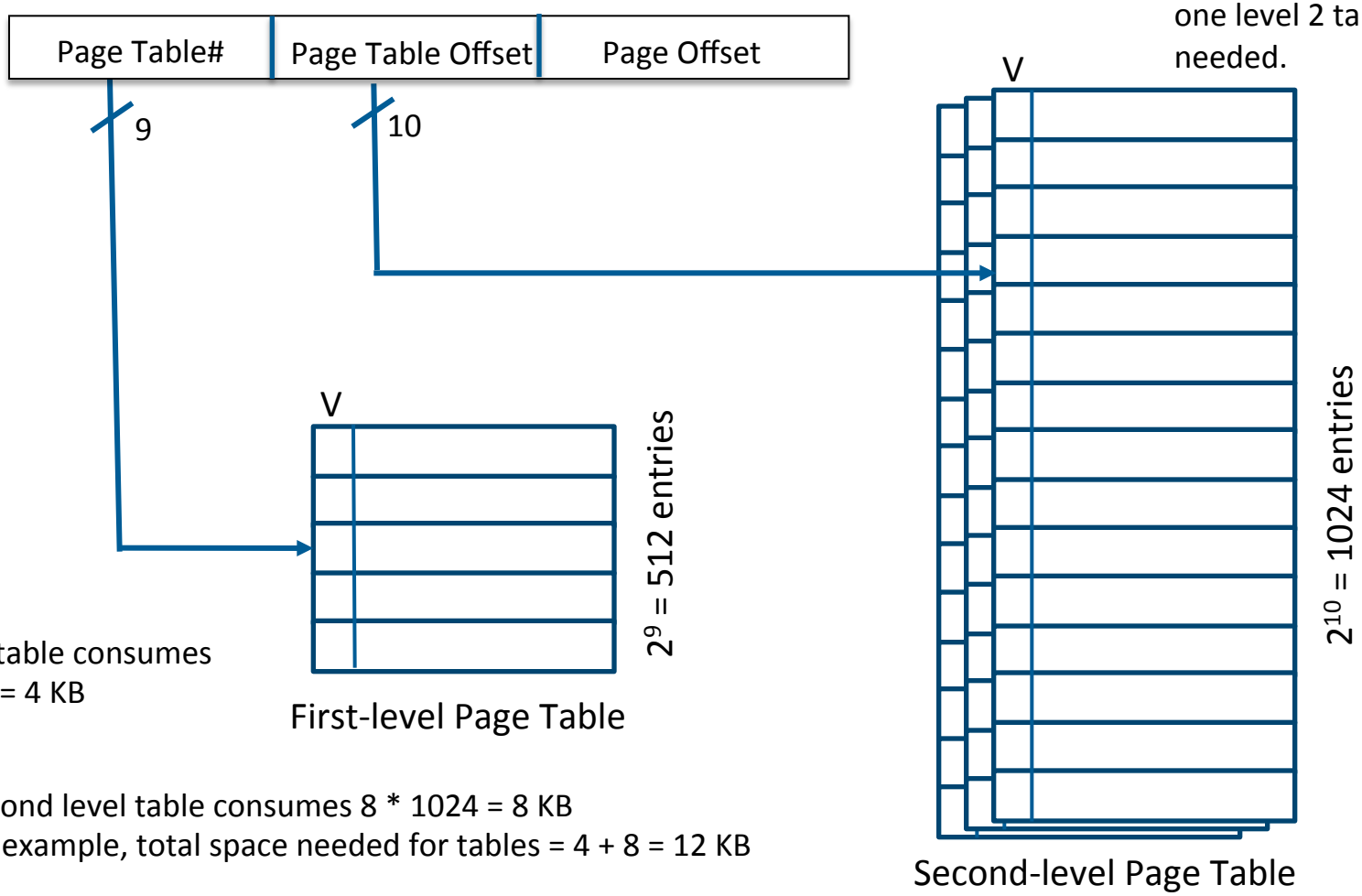
total PMT size = 8 * 524288 = 4 MB

Each program or process would have a separate 4 MB page map table.

Example 2-level System

The first level table is kept in memory.

If no more than 1024 pages are used, only one level 2 table is needed.



Level 1 table consumes
8 * 512 = 4 KB

One second level table consumes 8 * 1024 = 8 KB
For this example, total space needed for tables = 4 + 8 = 12 KB

Disadvantages of Multi-level PMT

- Multi-level PMTs increase the effective access time
  - *Each level requires a memory access*
  - *All but the first level could cause a page fault*

- Bits to left of offset must be split into N fields
  - One field for each of the levels
  - Virtual address translation takes longer

Inverted Page Map Table

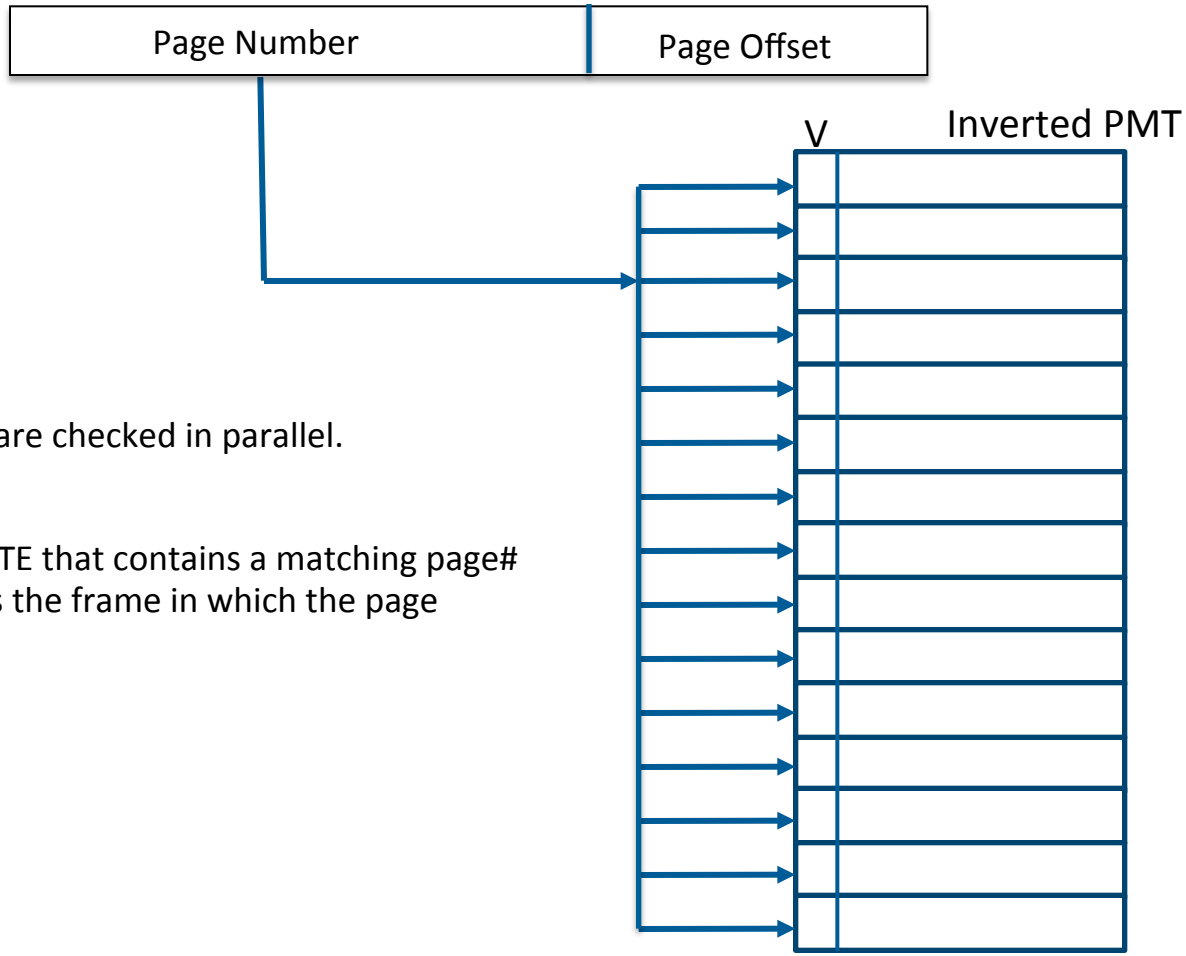
- PMTs described above, are "*forward*" page tables
- *Inverted* PMTs can reduce the required space
- A virtual address is divided into two fields:
  - *page* number field, and
  - *offset* field

Inverted Page Map Table

- There is one PTE per memory frame
  - Number of PTEs depends on physical memory size
  - If valid, the PTE identifies the page in the frame
  - Fault occurs if no PTE contains a matching page number
- There is one system wide inverted PMT
  - Instead of one PMT per process as with forward tables

Inverted Page Map Table

To translate address, one option is to perform associative search of the inverted PMT.



All PTEs are checked in parallel.

A valid PTE that contains a matching page# indicates the frame in which the page resides.

PTE index corresponds to frame number.

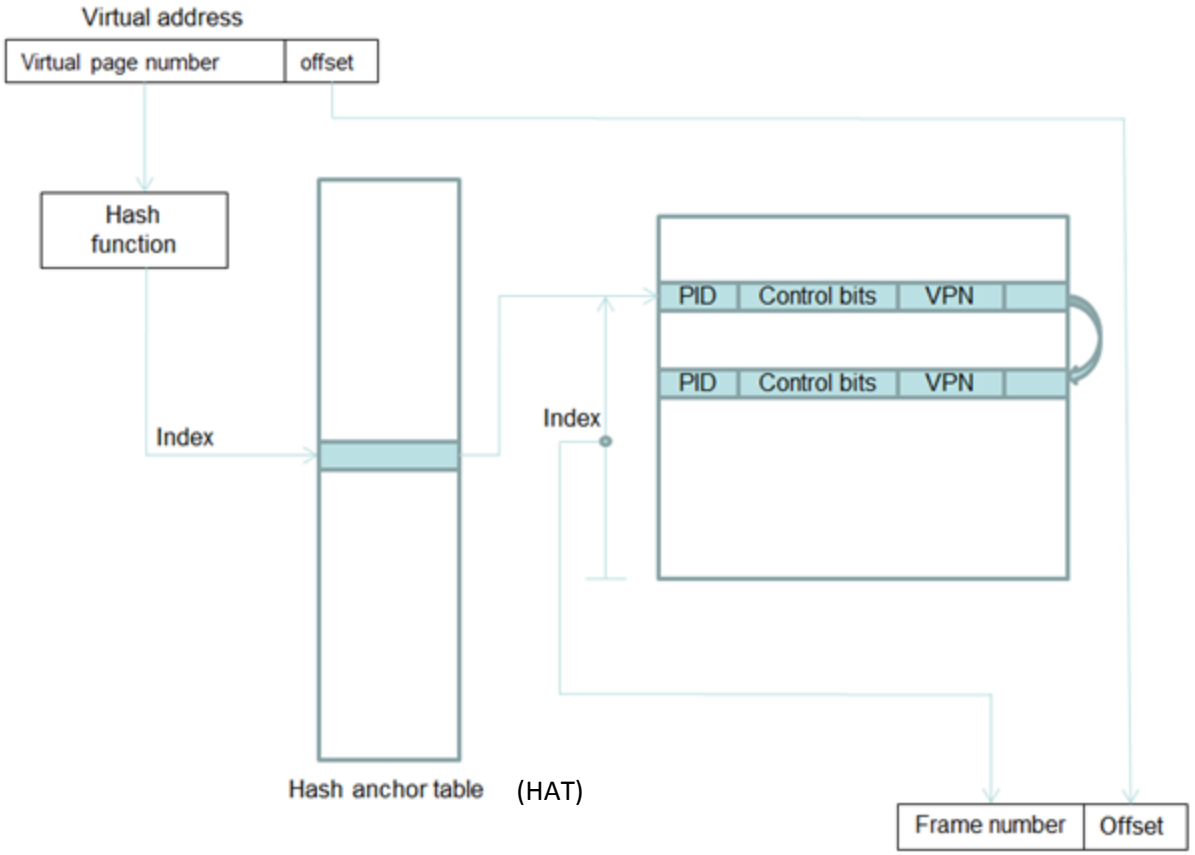
If physical memory contains M frames, the PMT will contain M PTEs.

Example: if entry 35 is valid and contains page# = 12, this means that frame 35 contains page 12.

Inverted Page Map Table

- There are alternatives to the associative PMT search
- Hash function may be used instead
  - *The page number is hashed into a table index*
  - *However, collisions must be handled*
  - *Happens if more than one page number hashes to same PTE*
- Collisions can be handled by using:
  - secondary hash function
    - Rehashes the page# into a different index
  - Chaining
    - Links entries that map to the same index

Inverted PMT Structure



First entry contains link to next entry resulting from collisions.

The HAT contains pointers to the heads of the chains for each page number. This allows more entries without large increases in the table size.