Cache Memory

Access Methods (1)

- Sequential
  - Start at the beginning and read through in order
  - Access time depends on location of data and previous location
  - e.g. tape
- Direct
  - Individual blocks have unique address
  - Access is by jumping to vicinity plus sequential search
  - Access time depends on location of data and previous location
  - e.g. disk

Access Methods (2)

- Random
  - —Individual addresses identify locations exactly
  - —Access time is independent of location or previous access
  - —e.g. RAM
- Associative
  - —Data is located by a comparison with contents of a portion of the store
  - —Access time is independent of location or previous access
  - —All memory is checked simultaneously; access time is constant
  - —e.g. cache

Performance

- From user's perspective the most important characteristics of memory are capacity and performance
- Three performance parameters:
  - —Access time
  - —Cycle Time
  - —Transfer Rate
- Access time (latency)
  - —For RAM, access time is the time between presenting an address to memory and getting the data on the bus
  - —For other memories, the largest component is positioning the read/write mechanism

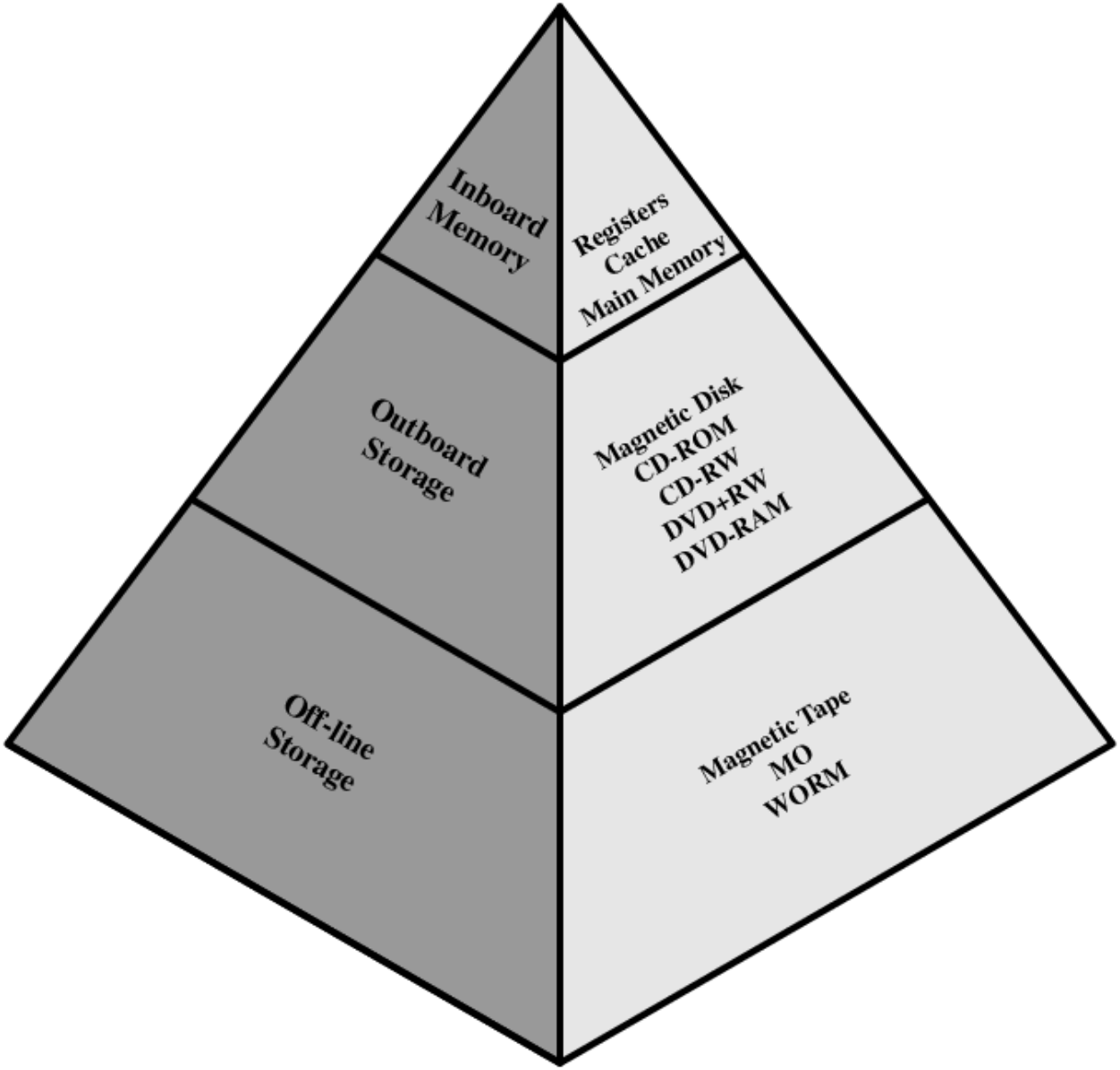
Performance

- Cycle Time
  - —Primarily applied to RAM; access time + additional time before a second access can start
  - —Function of memory components and system bus, not the processor
- Transfer Rate – the rate at which data can be transferred into or out of a memory unit
  - —For RAM, TR = 1 / (cycle time)
  - —**Transfer rate for other memories**
  - —Tn= Ta+ (n/r) where
  - —Tn=Average time to read or write N bits
  - —Ta=Average access time
  - —n = number of bits, r = transfer rate in bits / second

Memory Hierarchy

- For any memory:
  - How fast? - Time is money
  - How much? - Capacity
  - How expensive?
- Faster memory => greater cost per bit
- Greater capacity => smaller cost / bit
- Greater capacity => slower access
- Going down the hierarchy:
  - Decreasing cost / bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access by processor

Memory Hierarchy - Diagram



Reason That Cache Works: Locality of Reference

- Two or more levels of memory can be used to produce average access time approaching the highest level
- The reason that this works well is called "locality of references"
- In practice memory references (both instructions and data) tend to cluster
  - —Instructions: iterative loops and repetitive subroutine calls
  - —Data: tables, arrays, etc. Memory references cluster in short run

Principle of Locality

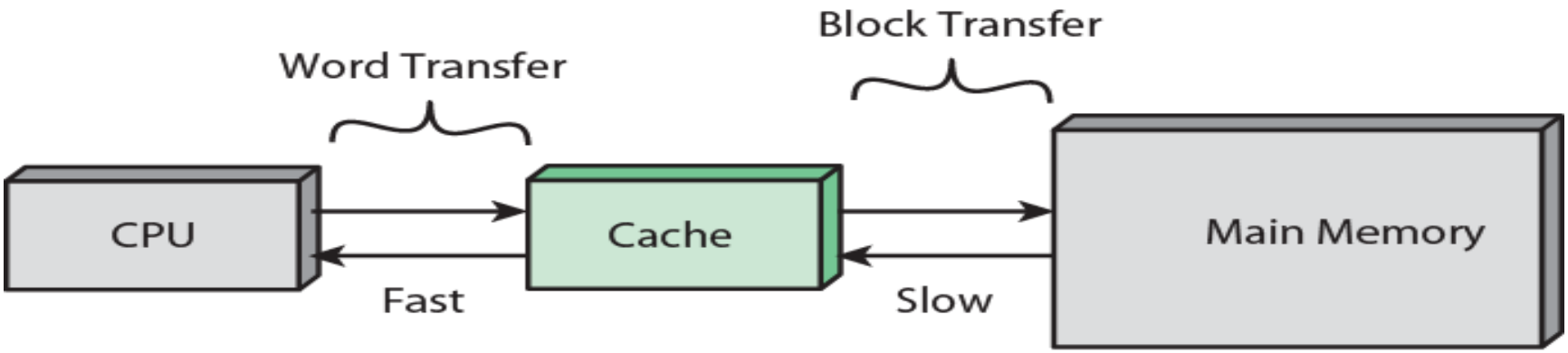
- Spatial Locality: Tendency for locations close to a location that has been accessed to also be accessed
- Temporal Locality: Tendency for a location that has been accessed to be accessed again
- Example

```
for(i=0; i<100000; i++)
    a[i] = b[i];
```

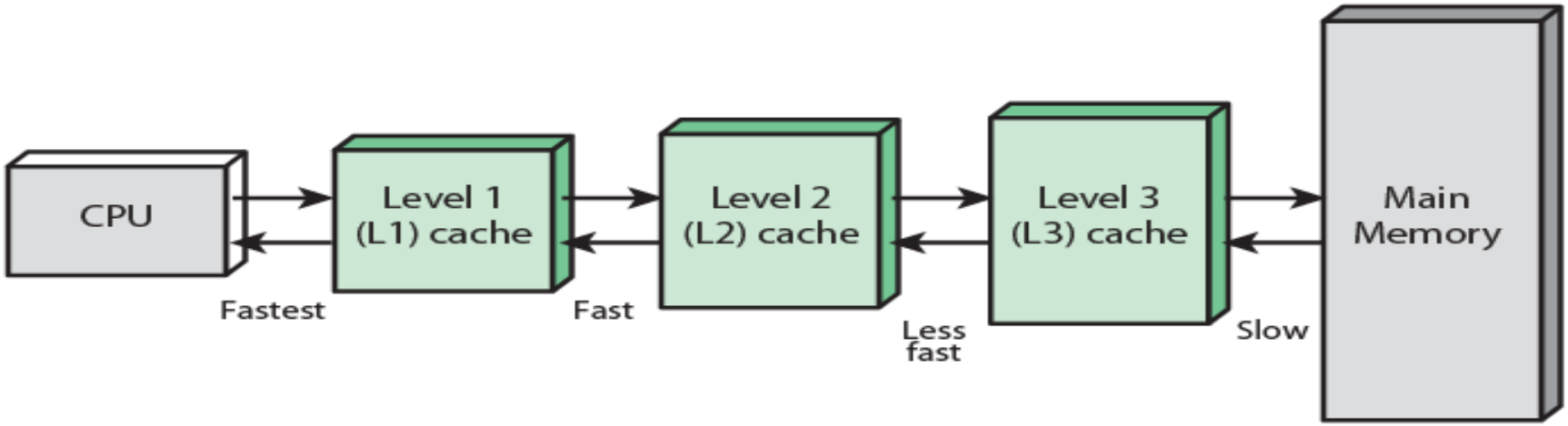
Cache

- A small amount of fast memory that sits between normal main memory and CPU
- May be located on CPU chip or module
- Intended to allow access speed approaching register speed
- When processor attempts to read a word from memory, cache is checked first
- **Cache Memory Principles:**
  - If data sought is not present in cache, a block of memory of fixed size is read into the cache
  - Locality of reference makes it likely that other words in the same block will be accessed soon

Cache and main memory



(a) Single cache



(b) Three-level cache organization

Cache operation - overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Basic terminologies

- ***Hit***: CPU finding contents of memory address in cache
- ***Hit rate*** *(h)* is probability of *successful lookup* in cache by CPU.
- ***Miss***: CPU *failing* to find what it wants in cache (incurs trip to deeper levels of memory hierarchy
- ***Miss rate*** *(m)* is probability of *missing* in cache and is equal to *1-h*.
- ***Miss penalty***: Time penalty associated with servicing a miss at any particular level of memory hierarchy
- ***Effective Memory Access Time (EMAT)***: Effective access time experienced by the CPU when accessing memory.
  - Time to lookup cache to see if memory location is already there
  - Upon cache miss, time to go to deeper levels of memory hierarchy

  $EMAT = Tc + m * Tm$

  where m is cache miss rate, Tc the cache access time and *Tm* the miss penalty

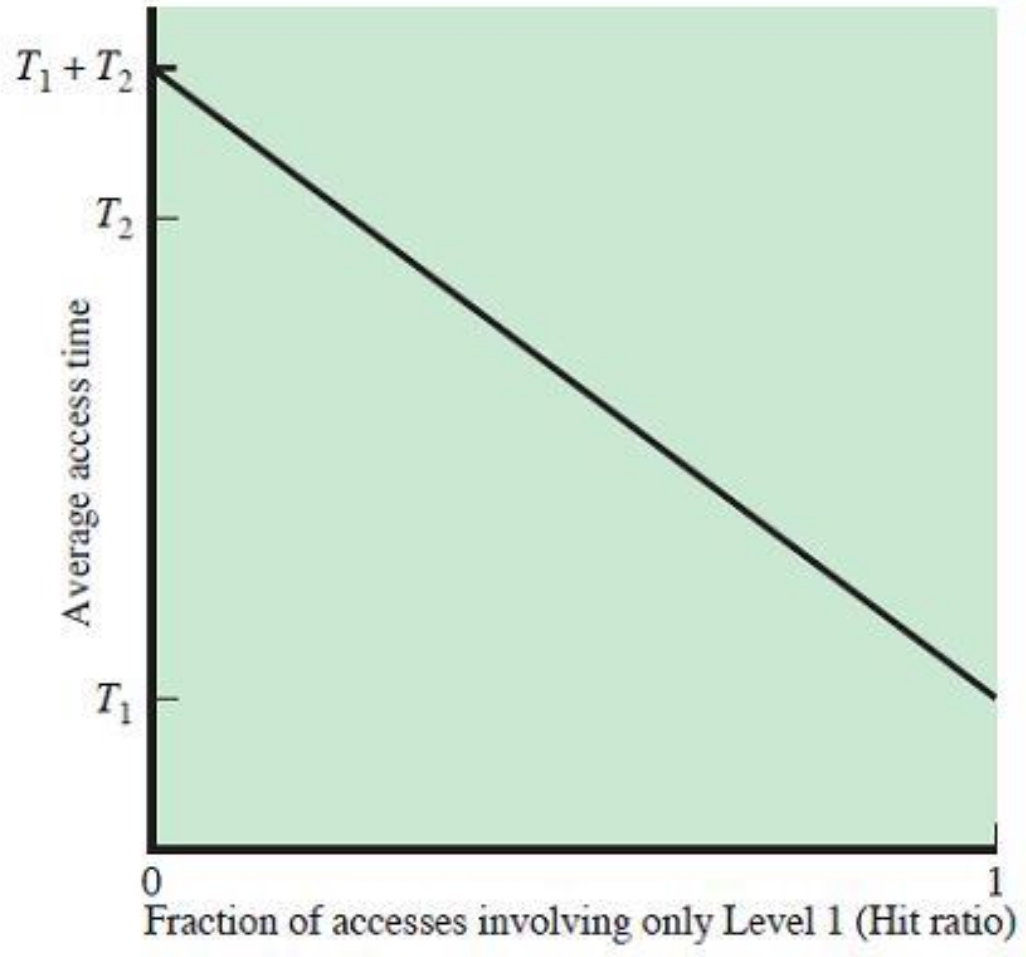
A Simple two-level cache

- Level 1: 1000 words, 0.01 microseconds
- Level 2: 100,000 words 0.10 microseconds

- If word in L1, processor has direct access else word copied from L2 into L1
- Avg Access Time as function of hit ratio H:

  H * 0.01 + (1-H)* 0.11
- With H near 1 access time approaches 0.01 microseconds

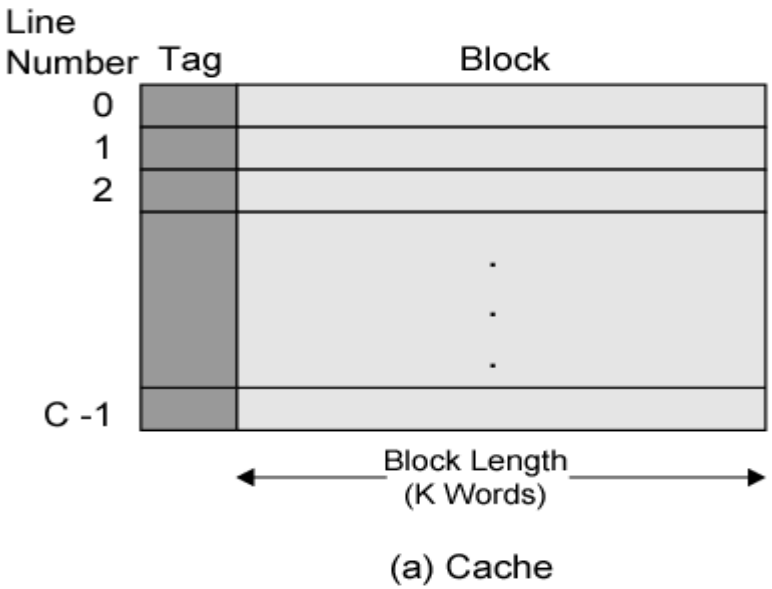
Two-level cache performance



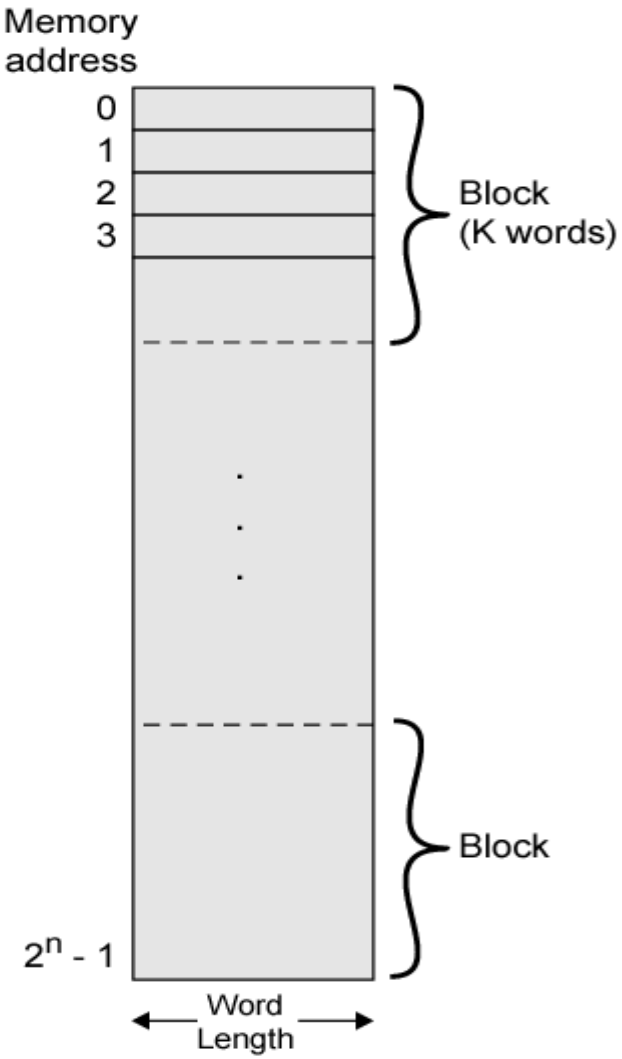
Two-level disk access

- Principles of two-level memories can be applied to disk as well as RAM
- A portion of main memory can be used as a disk cache
  - —Allows disk writes to be clustered; largest component of disk access time is seek time
  - —Dirty (modified) data may be requested by the program before it is even written back to disk

Cache/Main Memory Structure



(a) Cache

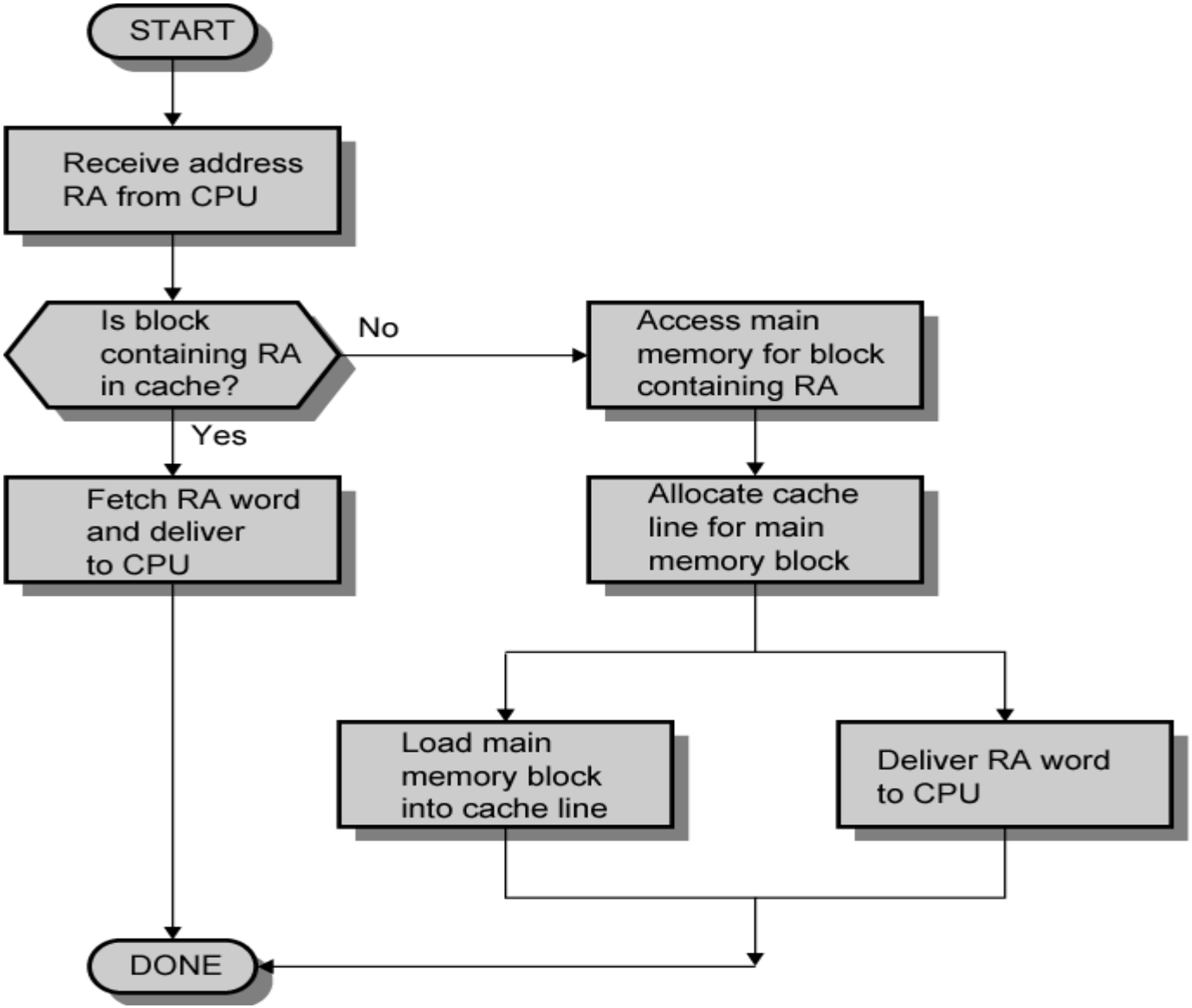


(b) Main memory

Cache view of memory

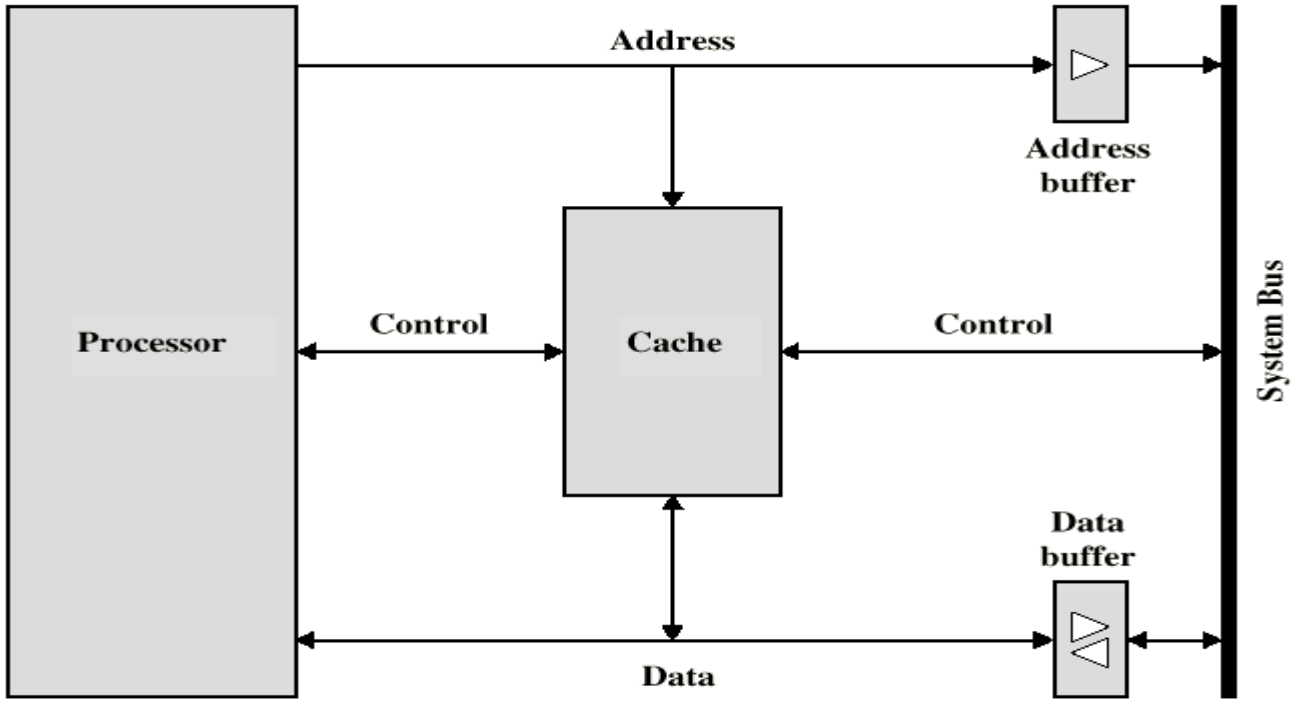
- n address lines => 2^n words of memory
- Cache stores fixed length blocks of K words
- Cache views memory as an array of M blocks where $M = 2^n/K$
- A block of memory in cache is referred to as a line. K is the line size
- Cache size is C blocks where C < M (considerably)
- Each line includes a tag that identifies the block being stored
- Tag is usually upper portion of memory address

Cache Read Operation - Flowchart



Typical Cache organization

- Illustrates a shared connection between processor, the cache and the system bus (look-aside cache)
- Another way to organize this system is to interpose the cache between the processor and the system bus for all lines (look-through cache)



Elements of Cache Design

- Size
- Mapping Function (direct, associative, set associative)
- Replacement Algorithm (LRU, LFU, FIFO, random)
- Write Policy (write through, write back, write once)
- Line Size
- Number of Caches (how many levels, unified or split)
  - Note that cache design for High Performance Computing (HPC) is very different from cache design for other computers
  - Some HPC applications perform poorly with typical cache designs

Cache Size does matter

- Cost
  - More cache is expensive
  - Would like cost/bit to approach cost of main memory
- Speed
  - But we want speed to approach cache speed for all memory access
  - More cache is faster (up to a point)
  - Checking cache for data takes time
  - Larger caches are slower to operate

Comparison of Cache Sizes

Processor	Type	Year of Introduction	L1 Cache[a]	L2 cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128 to 256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256 to 512 KB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 kB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 kB/32 kB	4 MB	—
Itanium 2	PC/server	2002	32 kB	256 KB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24-48 MB

Look-aside and Look-through

- Look-aside cache is parallel with main memory
- Cache and main memory both see the bus cycle
  - —Cache hit: processor loaded from cache, bus cycle terminates
  - —Cache miss: processor AND cache loaded from memory in parallel

- Pro: less expensive, better response to cache miss
- Con: Processor cannot access cache while another bus master accesses memory

Look-through cache

- Cache checked first when processor requests data from memory
  - Hit: data loaded from cache
  - Miss: cache loaded from memory, then processor loaded from cache
- Pro:
  - Processor can run on cache while another bus master uses the bus
- Con:
  - More expensive than look-aside, cache misses slower

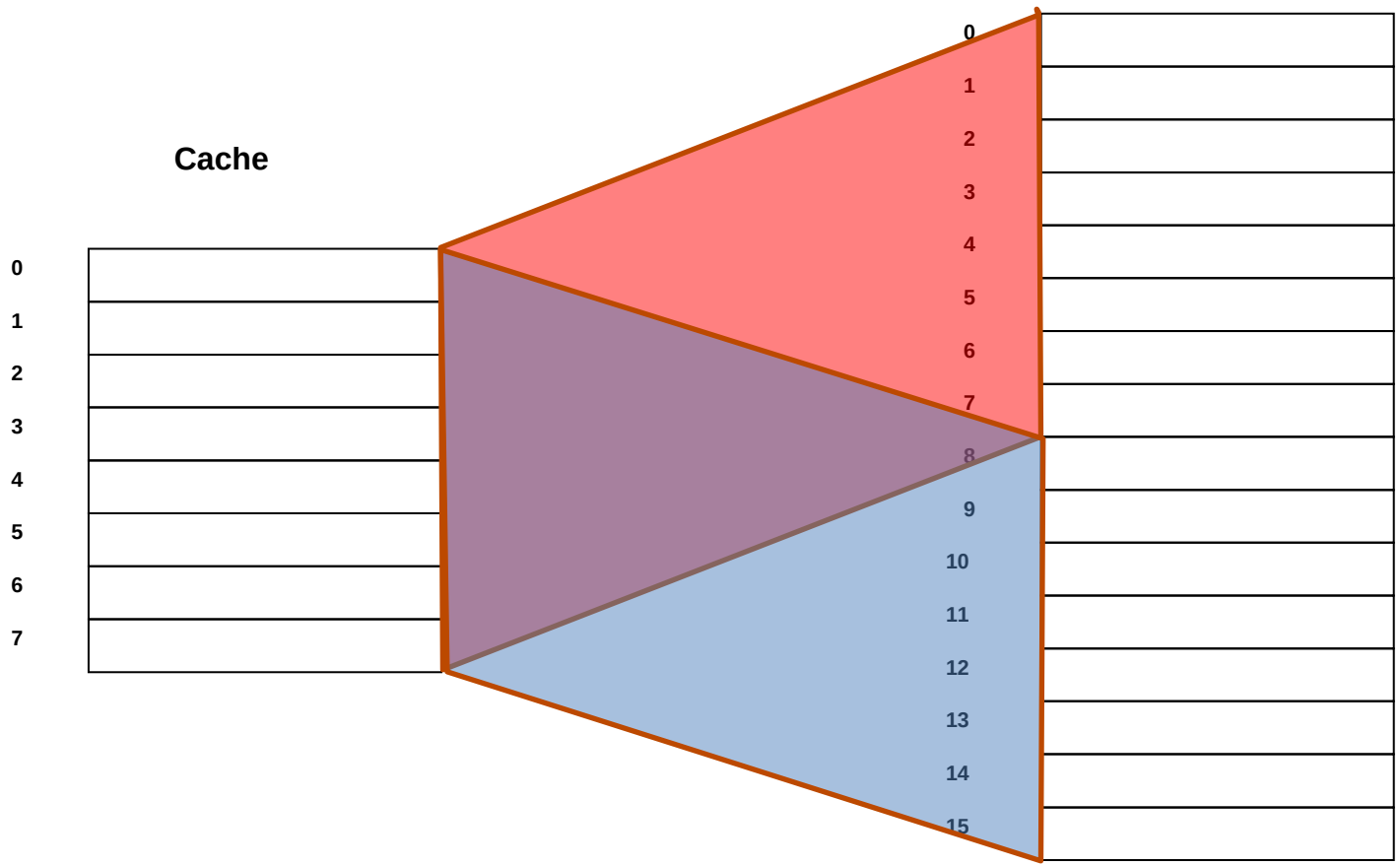
Mapping Functions

- There are fewer cache lines than memory blocks so we need
  - An algorithm for mapping memory into cache lines
  - A means to determine which memory block is in which cache line

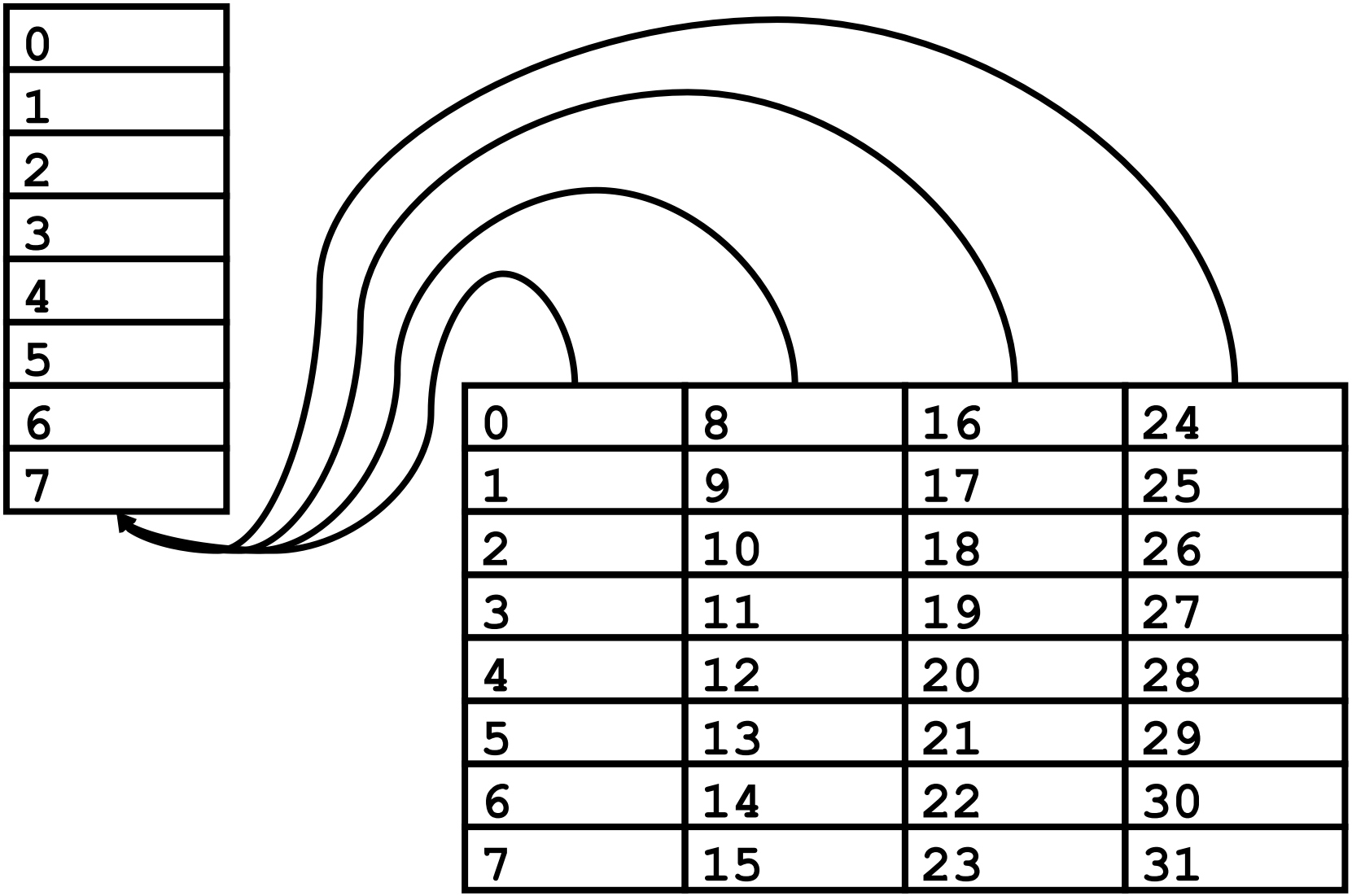
3 functions:

1) Direct Mapping
2) Associative Mapping
3) Set Associative Mapping

Direct-mapped cache organization



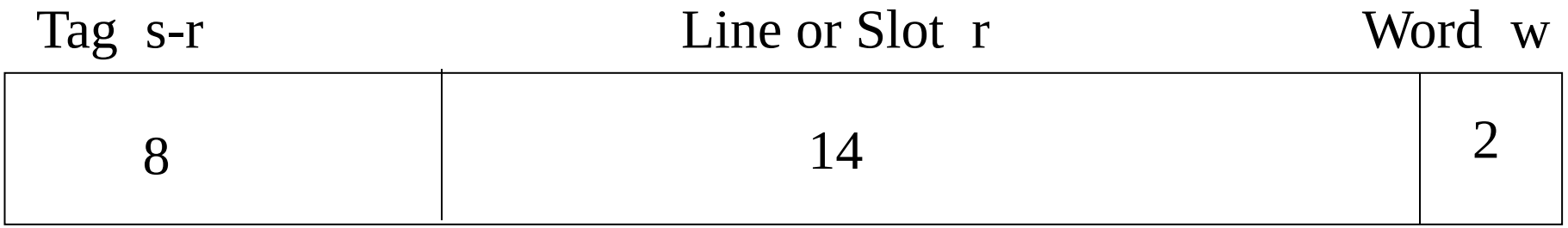
Direct-mapped cache organization



Direct Mapping

- Each block of main memory maps to only one cache line
  - i.e. if a block is in cache, it must be in one specific place
- Mapping function i = j modulo m (**i = j % m**) where
  - i = cache line number
  - j = main memory block number
  - m = number of cache lines
- Address is in two parts
- Least Significant w bits identify unique word
- Most Significant s bits specify one memory block
- The MSBs are split into a cache line field r and a tag of s-r (most significant)

Direct Mapping Address Structure

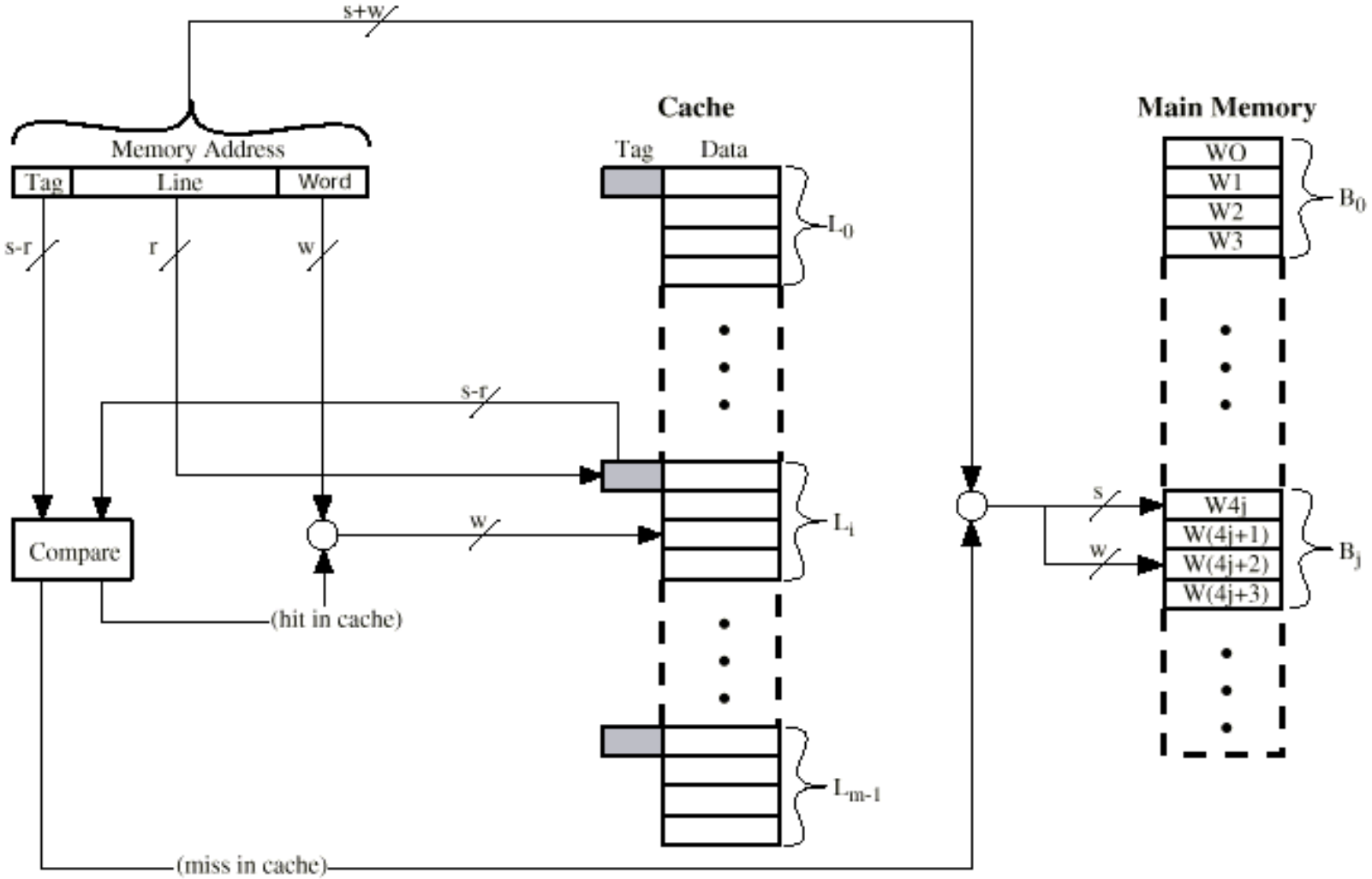


- 24 bit address, 2 bit word identifier (4 byte block)
- 22 bit block identifier (s)
  - 8 bit tag (=22-14) and 14 bit slot or line
  - Example: AB1402 tag=AB line=0500 word=2
  - Note: 1402 = 0001 0100 0000 0010
  - Remove l.s. 2 bits = 0001 0100 0000 00 = 00 0101 0000 0000 = 0500
- There are 2^s blocks in memory
- No two blocks with the same line number can have the same Tag field
  - AC1400, 041403, C71401 …
- Check contents of cache by finding line and checking Tag
  - Line is 0500 for all of these
  - If mem request is AB1402 tag at 0500 must = AB

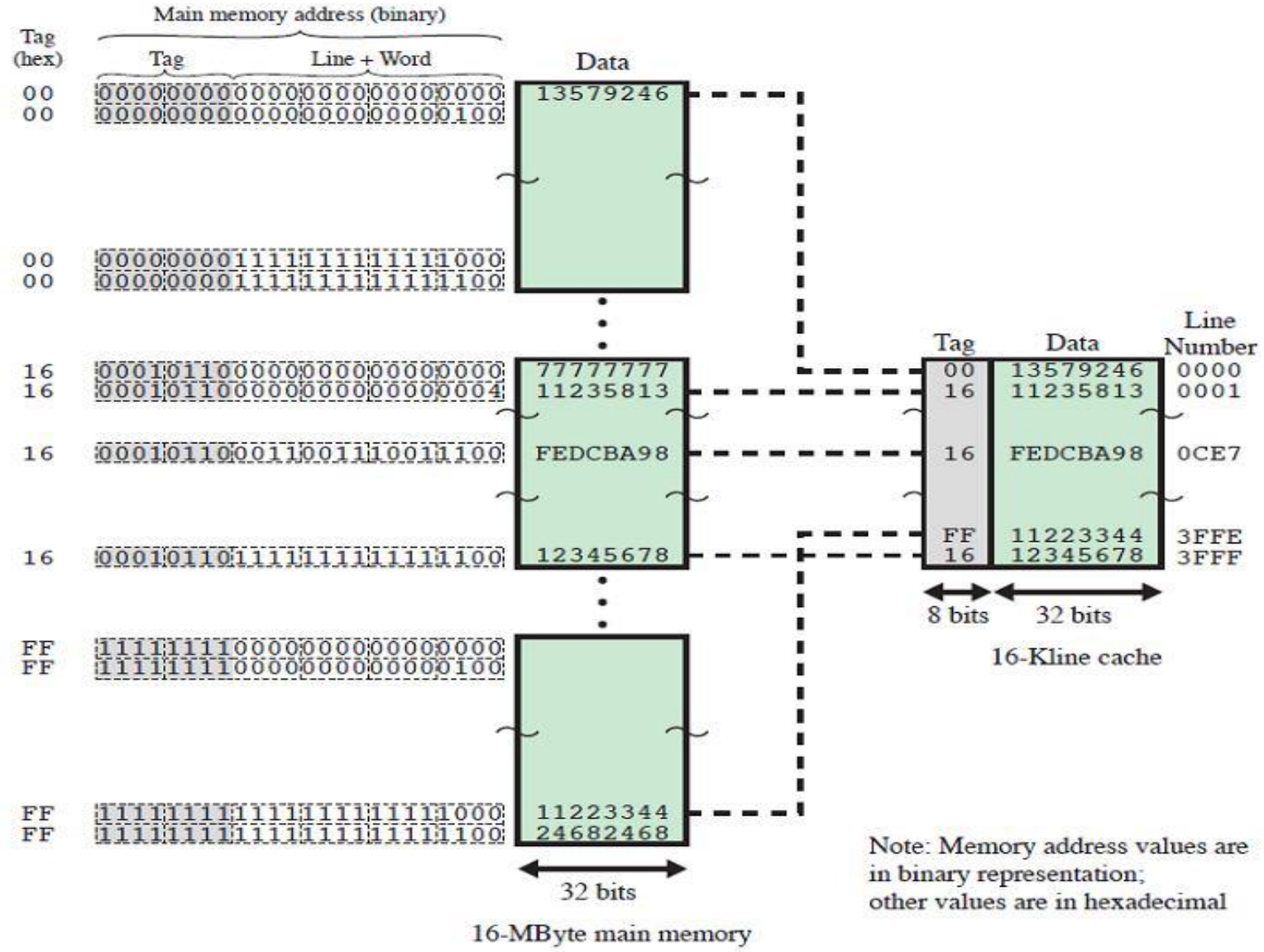
Direct Mapping

- Parking lot analogy: think of the cache as a parking lot, with spaces numbered 0000-9999
- With a 9 digit student id, we could assign parking spaces based on the middle 4 digits: xxx PPPP yy
- Easy to find your parking space
- Problem if another student is already there!
- Note that with memory addresses, the *middle* bits are used as a line number
  - Locality of reference suggests that memory references close in time will have the same highorder bits

Direct Mapping Cache Organization



Example


Main memory address (binary)
Tag (hex)
Tag
Line + Word
Data
00 0000000000000000000000000000
00 0000000000000000000000000100
00 0000000011111111111111111000
00 0000000011111111111111111100
16 0001011000000000000000000000
16 0001011000000000000000000004
16 00010110001100111001 1100
16 00010110111111111111 1100
FF 11111111000000000000000000000
FF 11111111000000000000000000100
FF 11111111111111111111 1111 1000
FF 11111111111111111111 1111 1100
77777777
11235813
FEDCBA98
12345678
11223344
24682468
13579246
32 bits
16-MByte main memory
Tag
Data
Line Number
00 13579246 0000
16 11235813 0001
16 FEDCBA98 0CE7
FF 11223344 3FFE
16 12345678 3FFF
8 bits
32 bits
16-Kline cache
Note: Memory address values are in binary representation; other values are in hexadecimal

Direct Mapping Summary

- Address length = (s + w) bits where w =
- $\log_2$(block size)
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $2^{s+w}/2^w = 2^s$
- Size of line field is r bits
  - Number of lines in cache = m = 2^r
  - Size of tag = (s – r) bits
- Size of cache 2^{r+w} bytes or words

Direct Mapping Cache Line Table

Cache line	Main Memory blocks held
0	0, m, 2m, 3m…2^s-m
1	1,m+1, 2m+1…2^s-m+1
m-1	m-1, 2m-1,3m-1…2^s-1

Direct Mapping Pros & Cons

- Pro
  - —Simple
  - —Inexpensive
- Con
  - —Fixed location for given block
  - —If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high (thrashing)
- Victim cache
  - —A solution to direct mapped cache thrashing
  - —Discarded lines are stored in a small “victim” cache (4 to 16 lines)
  - —Victim cache is fully associative and resides between L1 and next level of memory

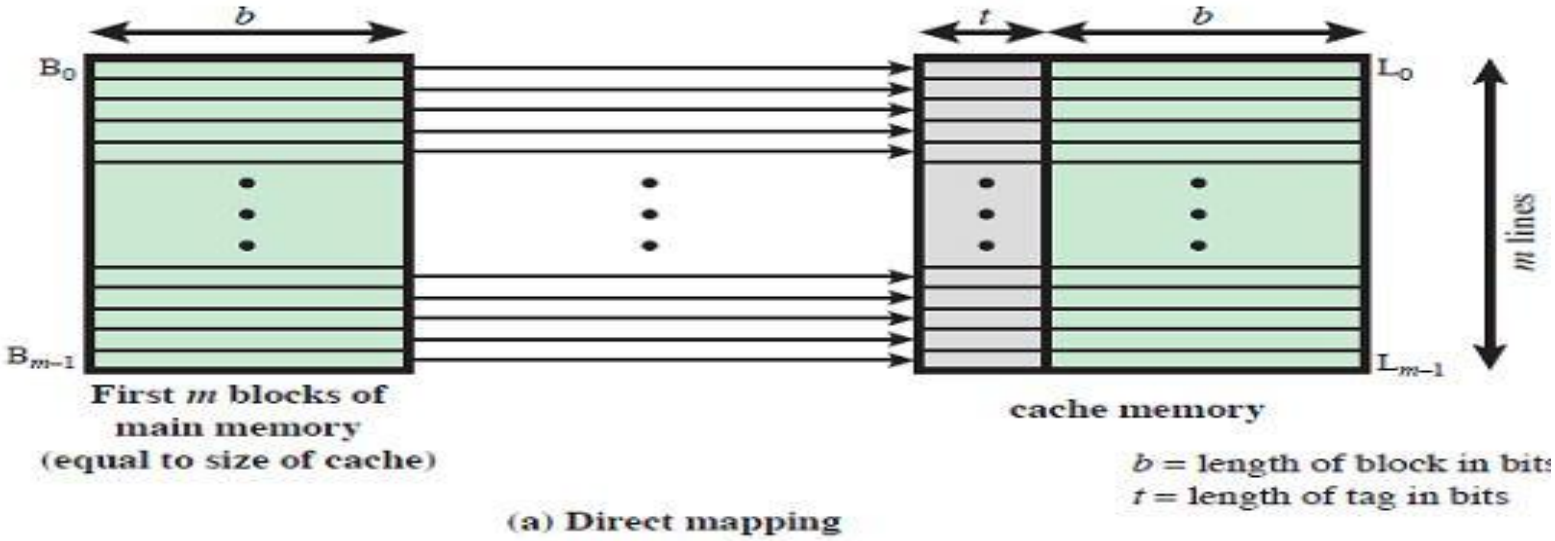
Associative Mapping

- A main memory block can load into any line of cache
- Memory address is interpreted as 2 fields: tag and word
- Tag uniquely identifies block of memory
- Every line's tag is examined simultaneously for a match
  - Cache searching gets expensive because a comparator must be wired to each tag
  - A comparator consists of XNOR gates (true when both inputs are true)
  - Complexity of comparator circuits makes fully associative cache expensive

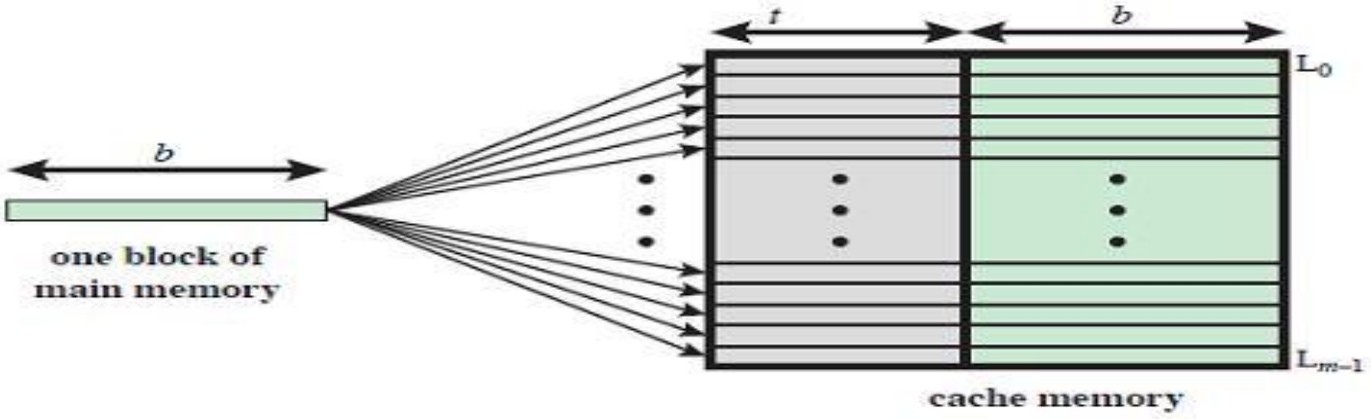
Associative Mapping

- Because no bit field in the address specifies a line number the cache size is not determined by the address size
- Associative-mapped memory is also called "content-addressable memory."
- Items are found not by their address but by their content
  - —Used extensively in routers and other network devices
  - —Corresponds to associative arrays in Perl and other languages
- Primary disadvantage is the cost of circuitry

Direct Mapping compared to Associative



(a) Direct mapping



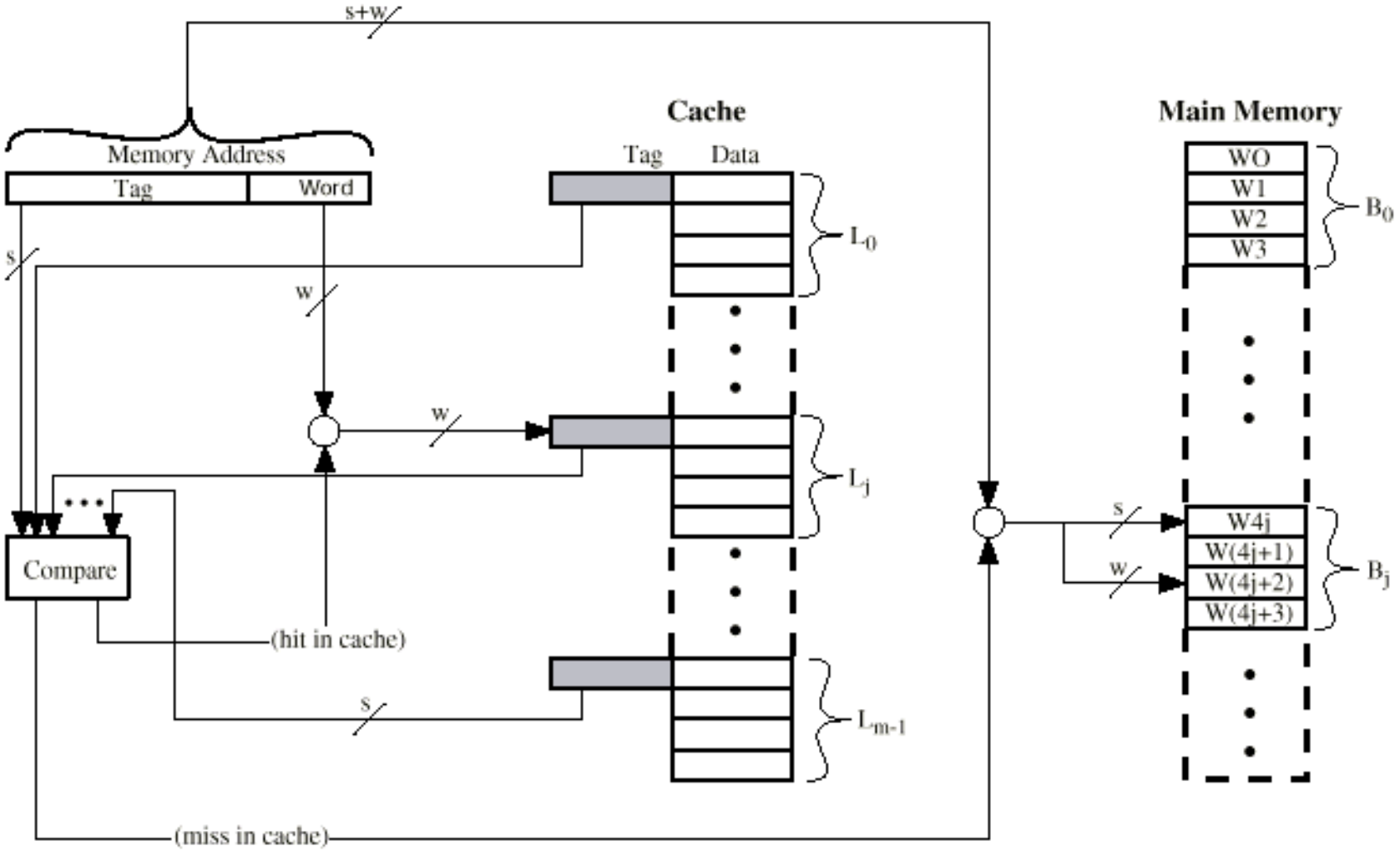
(b) Associative mapping

Associative Mapping Address Structure

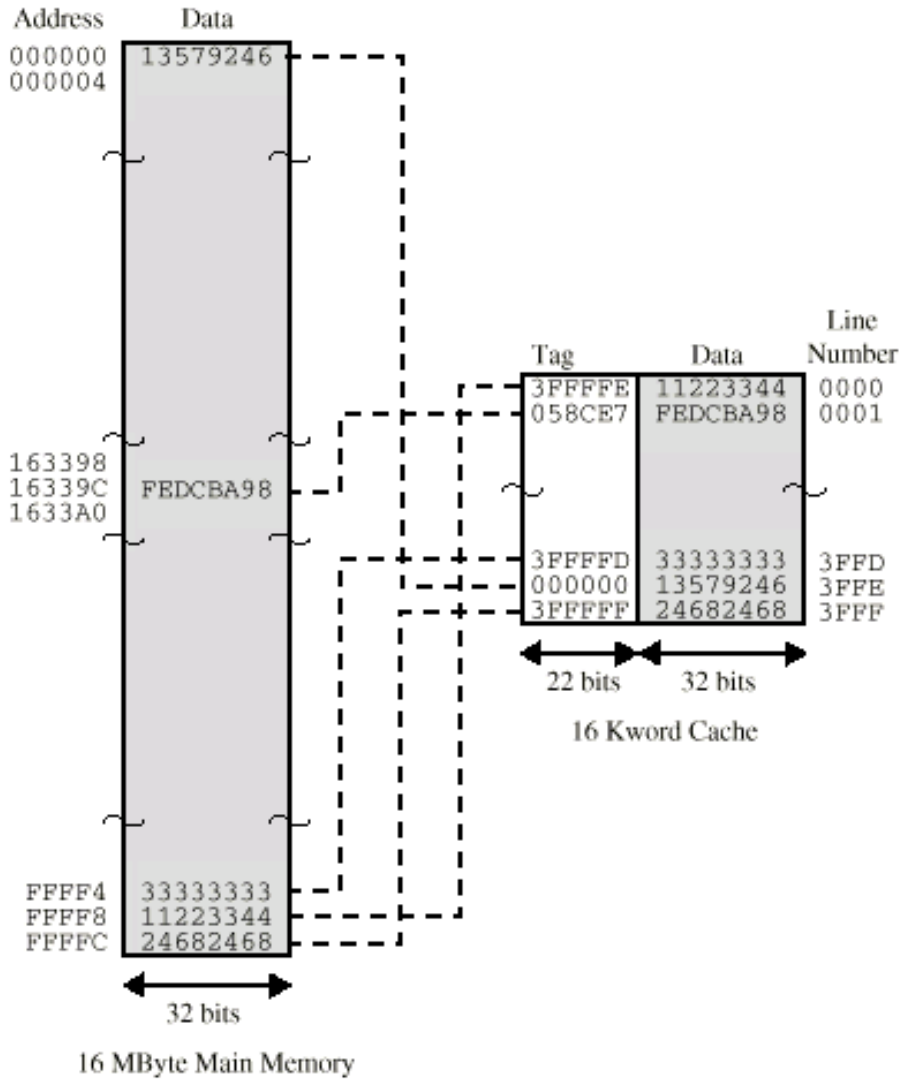
- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block
- e.g.

—Address	Tag	Data	Cache line
—FFFFFC	FFFFFC	24682468	3FFF

Fully Associative Cache Organization



Associative Mapping Example



Associative Mapping

- Parking lot analogy: there are more permits than spaces
- Any student can park in any space
- Makes full use of parking lot
  - —With direct mapping many spaces may be unfilled
- Note that associative mapping allows flexibility in choice of replacement blocks when cache is full
- Discussed below

Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $2^{s+ w}/2^{w} = 2^{s}$
- Number of lines in cache = undetermined
- Size of tag = s bits

Set Associative Mapping

- A compromise that provides strengths of both direct and associative approaches
- Cache is divided into a number of sets of lines
- Each set contains a fixed number of lines
- A given block maps to any line in a *given set* determined by that block's address
  - —e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - —2 way associative mapping
  - —A given block can be in one of 2 lines in only one set

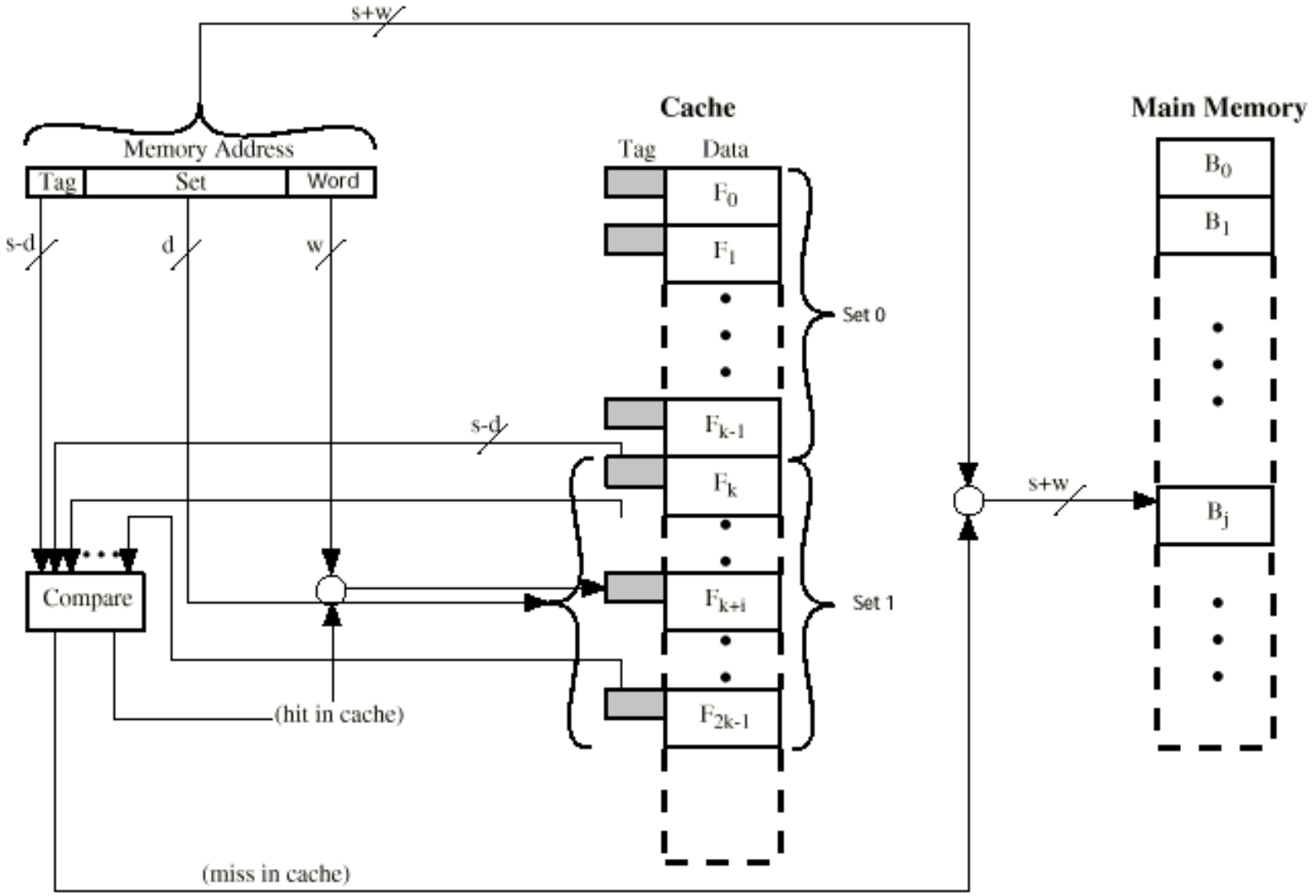
Set Associative Mapping

- m = v * k
  - Where m = number of lines in cache, v = number of sets and k = lines/set
  - Lines in cache = sets * lines per set
- i = j modulo v
  - Where i = set number and j = main memory block number
  - Set number = block number % number of sets
- This is referred to as a "k-way" set associative mapping
- Block Bi can be mapped only into lines of set j.

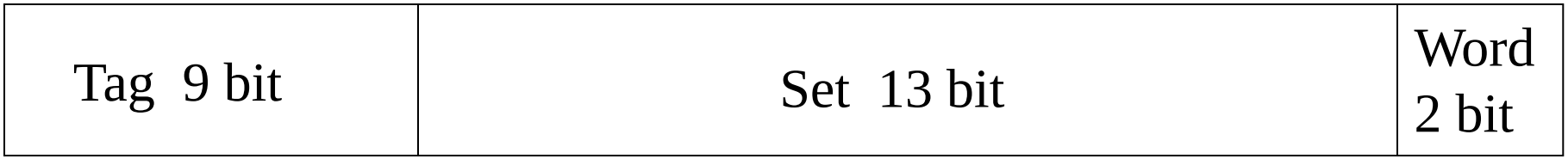
Set Associative Mapping: Parking Analogy

- If we have 10,000 parking spaces we can divide them into 1000 sets of 10 spaces each
- Still use middle digits of id to find your parking place *set:* xxx PPP yyy
- You have a choice of any place in your set
- Our parking lots actually work like this, but the sets are fairly large: Fac/Staff; Commuter; Resident; Visitor

Two Way Set Associative Cache Organization

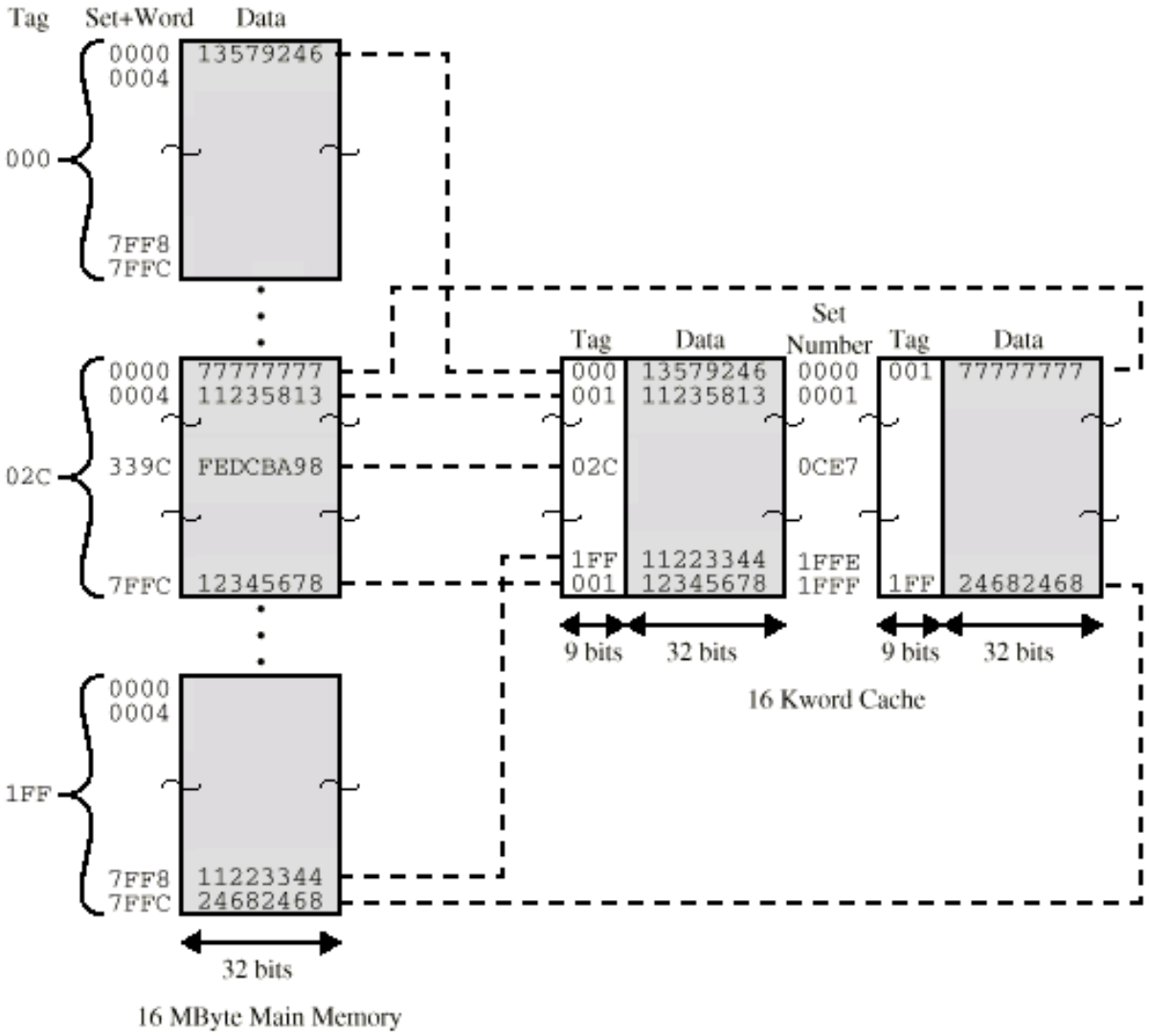


Set Associative Mapping Address Structure



- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- e.g
  - Address Tag Data Set number
  - 1FF 7FFC 1FF 12345678 1FFF
  - 001 7FFC 001 11223344 1FFF

Two Way Set Associative Mapping Example



Set Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = 2^{d}
- Number of lines in set = k
- Number of sets = v = 2^{d}
- Number of lines in cache = kv = k * 2^{d}
- Size of tag = (s – d) bits

Cache replacement algorithms

- When a line is read from memory it replaces some other line already in cache
- Other than direct mapping, there are choices for replacement algorithm
- Any given choice can result in a great speedup for one program and slow-down for some other program
- There is no "best choice" that works for all programs

- **Direct mapping-Replacement Algorithm**
  - —No choice, each block only maps to one line
  - —Replace that line

Replacement Algorithms for Associative & Set Associative

- Hardware implemented algorithm for speed
- Least Recently used(LRU) assumes locality of reference so *most* recently used is likely to be used again
- e.g. in 2 way set associative
  - —Which of the 2 block is lru?
- First in first out (FIFO)
  - —replace block that has been in cache longest
- Least frequently used
  - —replace block which has had fewest hits
- Random
- LRU is often favored because of ease of hardware implementation

Write Policy

- When a block of memory about to be overwritten in cache:
  - —No problem if not modified in cache
  - —Has to written back to main memory if modified (dirty)
- Must not overwrite a cache block unless main memory is up to date
- **Problems with dirty memory**
  - —More than one device may have access to main memory
    - I/O may address main memory directly
    - If word altered in cache, then main memory is invalid
    - If word altered in memory, then cache is invalid
  - —Multiple CPUs may have individual caches
    - Word altered in one cache may invalidate other caches

Write through vs. Write back

- **Write through**
  - Simplest technique
  - All writes go to main memory as well as cache
  - Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date
  - Lots of traffic, slows down writes
- **Write back**
  - Updates initially made in cache only
  - Update bit for cache slot is set when update occurs
  - If block is to be replaced, write to main memory only if update bit is set
  - Other caches get out of sync
  - I/O must access main memory through cache
  - 15% of memory references are writes

Cache Coherency (More in Ch. 18)

- In a bus organization with shared memory and multiple caches, coherency has to be maintained between caches as well as cache and memory
- Possible approaches:
  1. Bus watching with write through. Cache controller monitors bus lines and detects writes to memory in cache. Requires write-through policy for ALL cache controllers
  2. Hardware transparency. Extra hardware ensures that a write to one cache updates memory and all other caches
  3. Noncacheable memory. Memory shared between processors is designated as non-cacheable. All accesses to shared memory are cache misses. Mem identified with chip-select logic or high address bits

Line Size

- When a cache line is filled it normally includes more than the requested data– some adjacent words are retrieved
- As block size increases, cache hit ratio will also increase because of locality of reference–to a limit
- If block size is too large, possibility of reference to parts of block decreases; there are fewer blocks in cache so more chance of block being overwritten
- Relationship between block size and hit ratio is complex and program-dependent
- No optimal formula exists, but general purpose computing uses blocks of 8 to 64 bytes
- In HPC 64 and 128 byte lines are most common

Number of caches: multilevel caches

- With increased logic density, caches can be on same chip as cpu
- Reduces external bus activity and speeds up execution times
- No bus cycles; shorter data path is faster than 0-wait bus cycles
- Bus is free to do other transfers
- It is usually desirable to have external as well as internal cache
- With only 1 level bus access to memory is slow
- Most contemporary computers have at least 2 levels
  - Internal: Level 1 (L1)
  - External: Level 2 (L2)
- External L2 cache typically built with fast SRAM; uses separate and faster data bus, now incorporated on processor chip
- L3 Cache: performance improvements depend on hit rates, complicates replacement algorithms and write policy
- With L2 cache on-board, L3 cache can improve performance just as L2 can improve over L1 alone

Unified and Split Caches

- Split caches have separate caches for instructions and data
  - —These tend to be stored in different areas of memory
- Pros of unified cache:
  - —Higher rate for given cache size because cache is automatically balanced between instructions and data
  - —Only one cache needs to implemented
- **Split Cache:**
- Current trend favors split caches
  - —Useful for superscalar machines with parallel execution of instructions and prefetching of predicted instructions
  - —Split cache eliminates contention for cache between instruction fetch/decode unit and the execution unit (when accessing data)
  - —Helps to keep pipeline full because the EU will block the fetch/decode unit otherwise