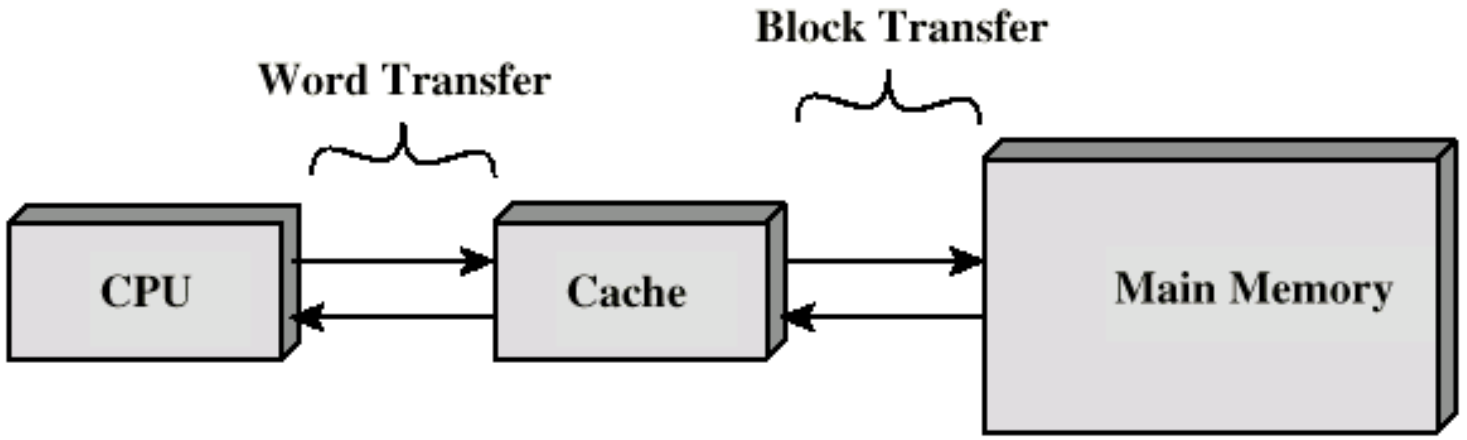
CACHE MEMORY

Cache Memory

- Small amount of fast memory, expensive memory
- Sits between normal main memory (slower) and CPU
- May be located on CPU chip or module
- It keeps a copy of the most frequently used data from the main memory.
- Reads and writes to the most frequently used addresses will be serviced by the cache.
- We only need to access the slower main memory for less frequently used data.



Principle of Locality

- In practice, most programs exhibit *locality, which the cache can take* advantage of.
- The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
- The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Temporal Locality in Programs and Data

- **Programs:** Loops are excellent examples of temporal locality in programs.
  - The loop body will be executed many times.
  - The computer will need to access those same few locations of the instruction memory repeatedly.
- **Data:** Programs often access the same variables over and over, especially within loops.

Spatial Locality in Programs and Data

- **Programs:** Nearly every program exhibits spatial locality, because instructions are usually executed in sequence—if we execute an instruction at memory location *i, then we will probably also execute the next instruction, at* memory location *i+1.*
- Code fragments such as loops exhibit *both temporal and spatial locality.*
- **Data:** Programs often access data that is stored contiguously.
  - Arrays, like a in the code on the top, are stored in memory contiguously.
  - The individual fields of a record or object like employee are also kept contiguously in memory.

How caches take advantage of temporal locality

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache *instead of* accessing the slower dynamic memory.
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.

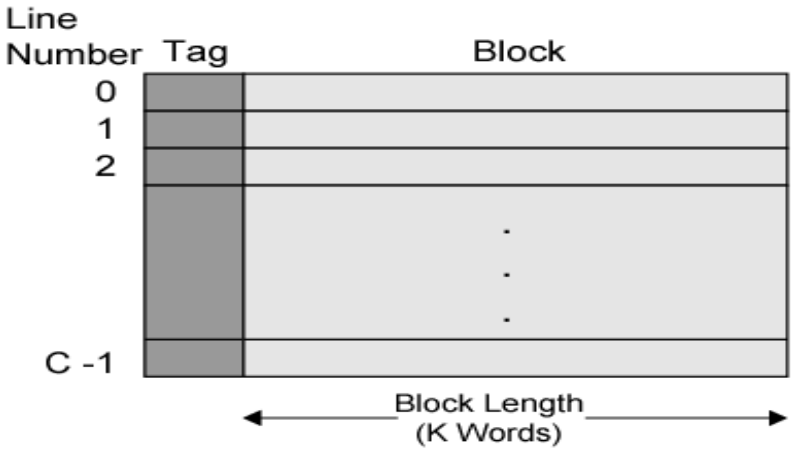
How caches take advantage of spatial locality

- When the CPU reads location *i from main memory, a* copy of that data is placed in the cache.
- But instead of just copying the contents of location *i,* we can copy *several values into the cache at once,* such as the four bytes from locations *i through i + 3.*
  - If the CPU later does need to read from locations *i + 1, i + 2 or i + 3, it can access that data from* the cache and not the slower main memory.
  - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data.

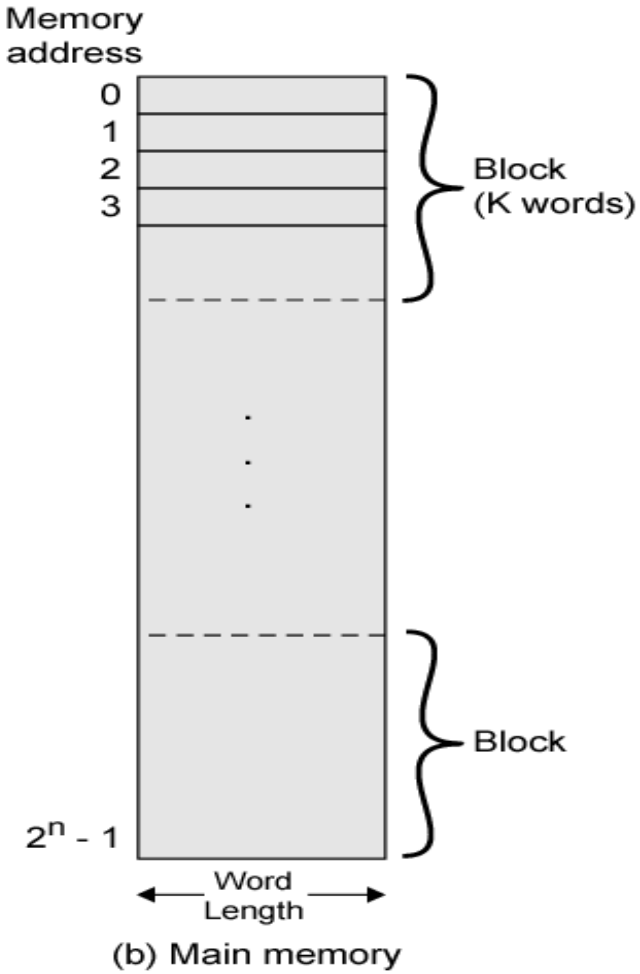
Hits and Misses

- A cache hit occurs if the cache contains the data that we're looking for.
  - cache can return the data much faster than main memory.
- A cache miss occurs if the cache does not contain the requested data.
  - CPU must then wait for the slower main memory.

Cache/Main Memory Structure



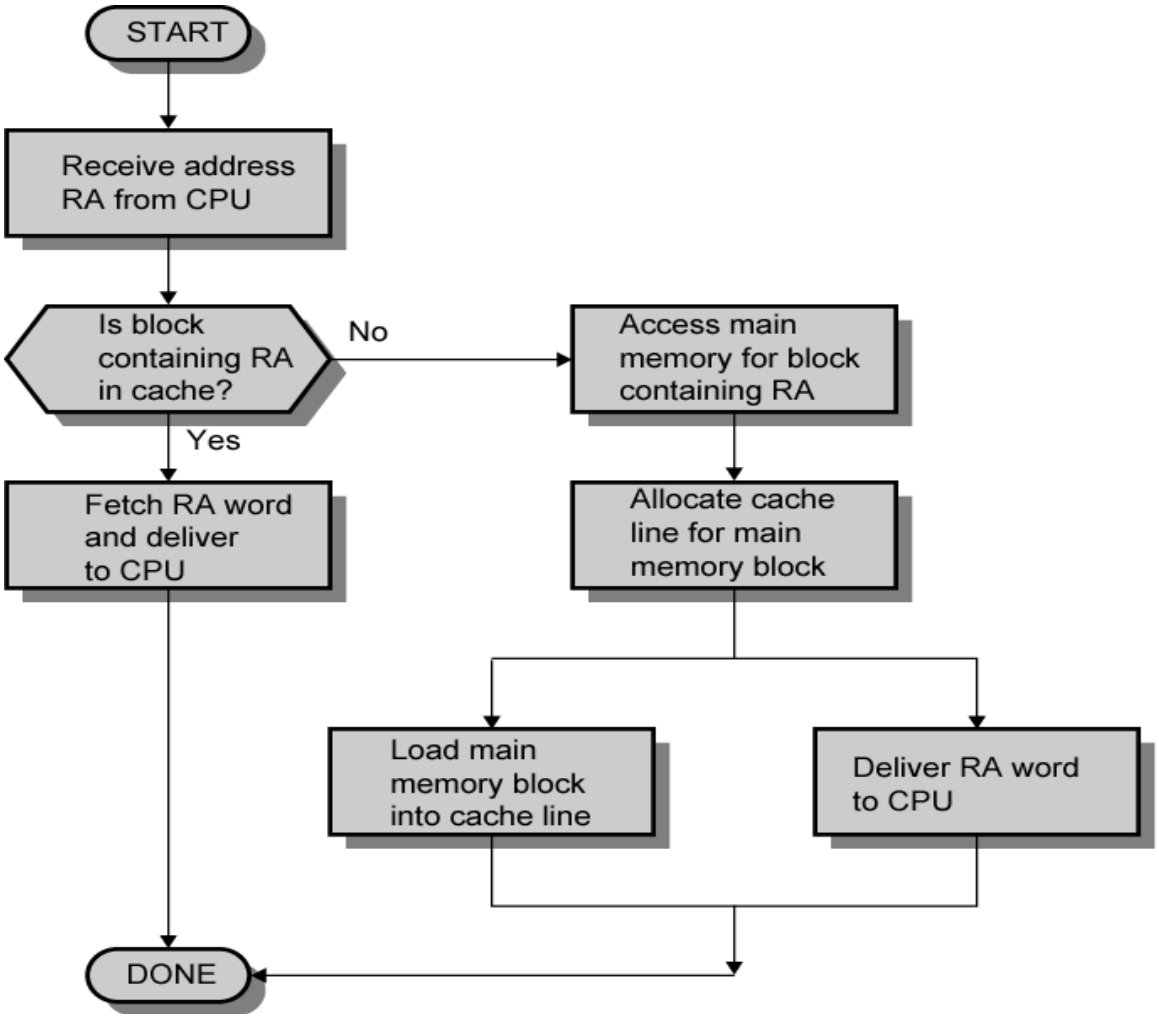
(a) Cache



(b) Main memory

Cache read operation

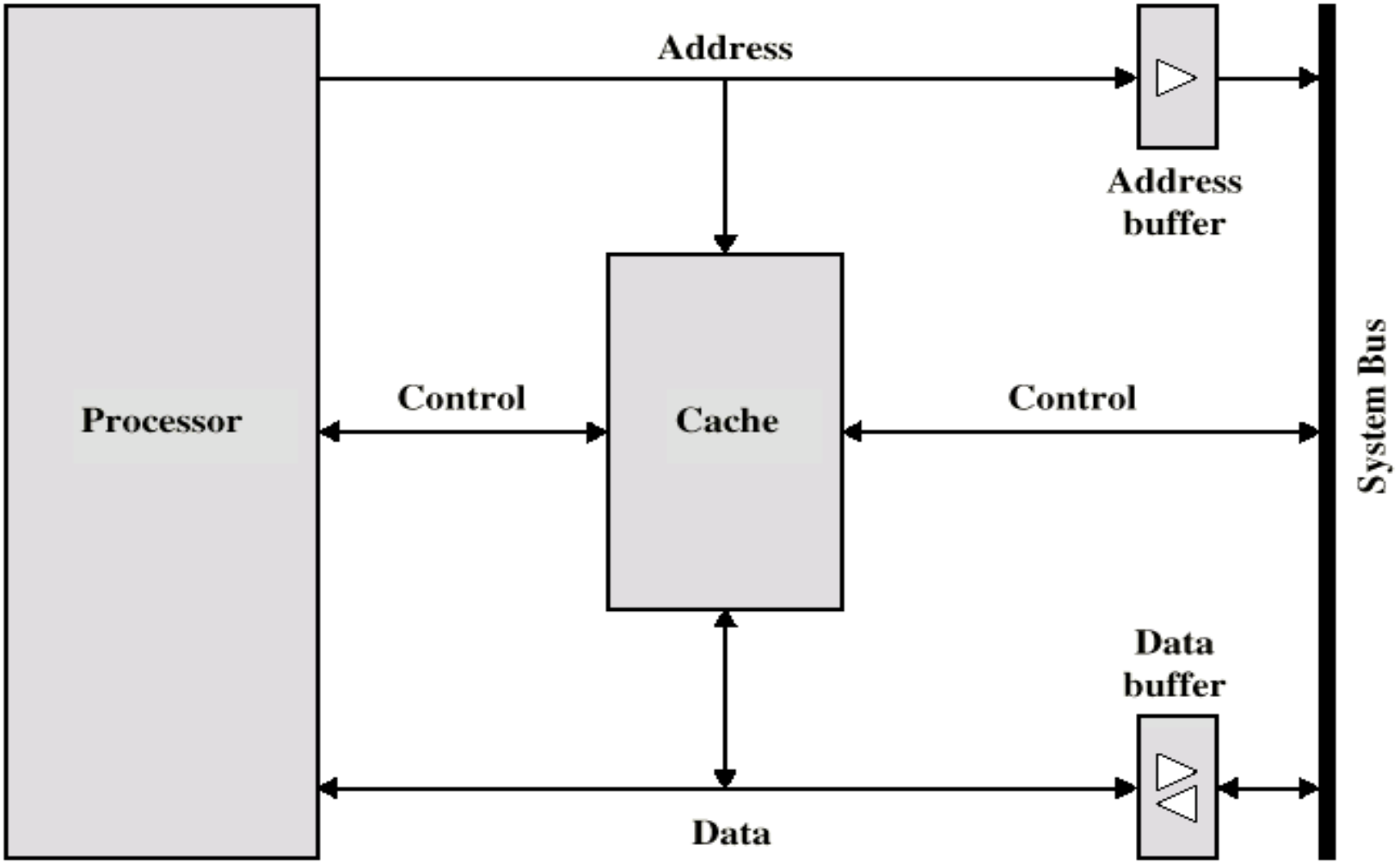
- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot



Size of Cache Design

- Cost
  - More cache is expensive
- Speed
  - More cache is faster (up to a point)
  - Checking cache for data takes time

Typical Cache Organization

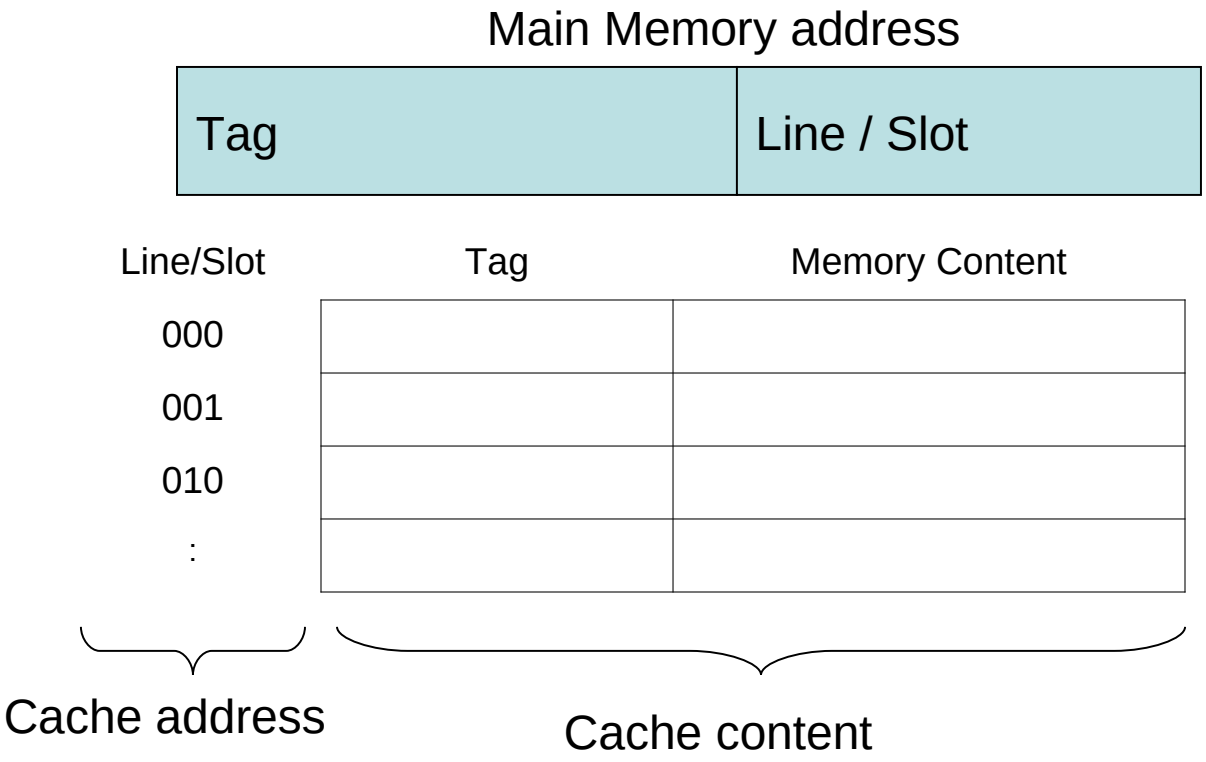


Comparison of Cache Sizes

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 MTAb	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	—

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
- Address is in two parts



Example 1: Direct Mapping

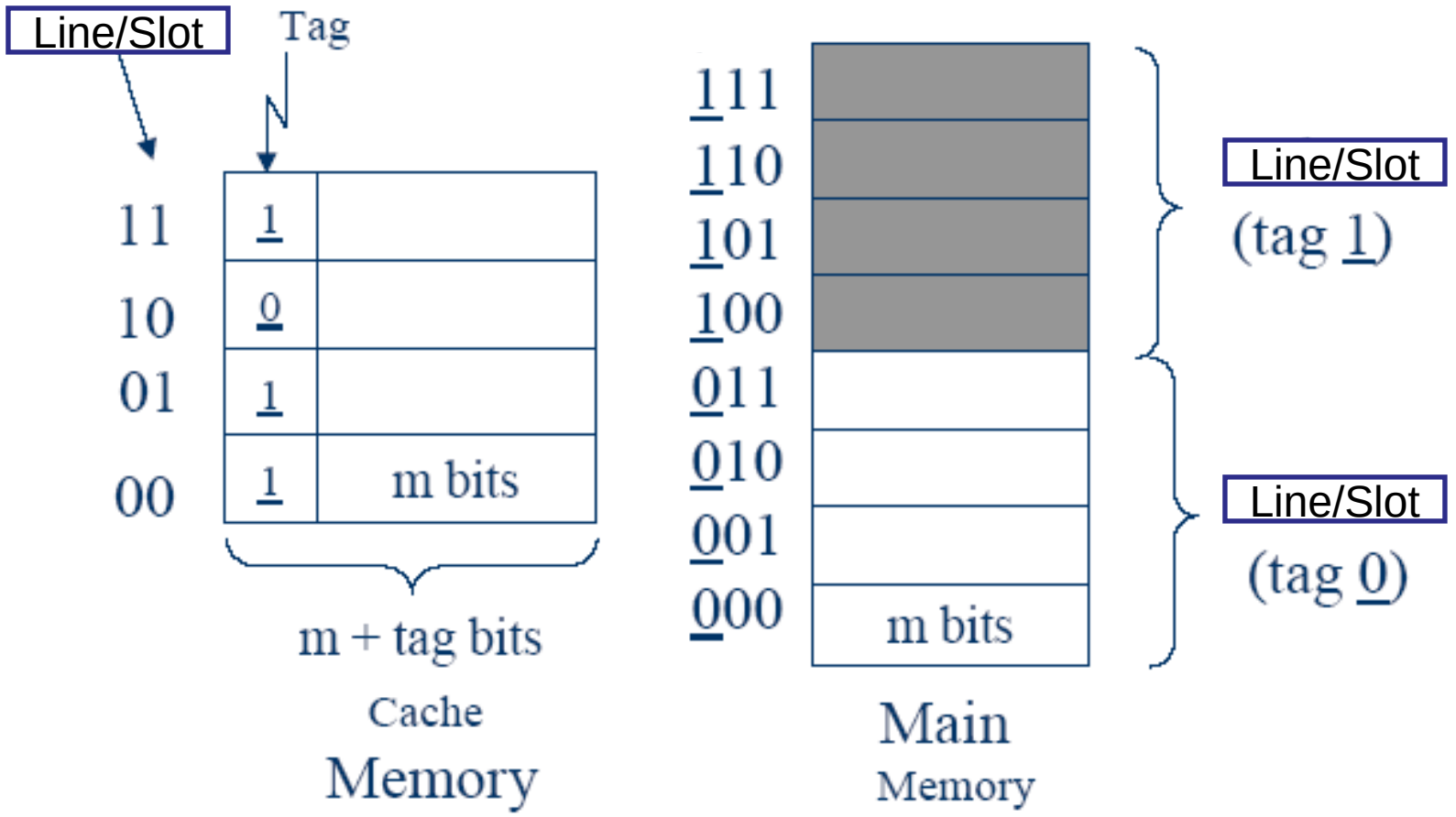
A main memory contains 8 words while the cache has only 4 words. Using direct address mapping, identify the fields of the main memory address for the cache mapping.

Solution:
Total memory words = 8 = 2^3 → Require 3 bit for main memory address.
Total cache words = 4 = 2^2 → Require 2 bit for cache address → line / slot

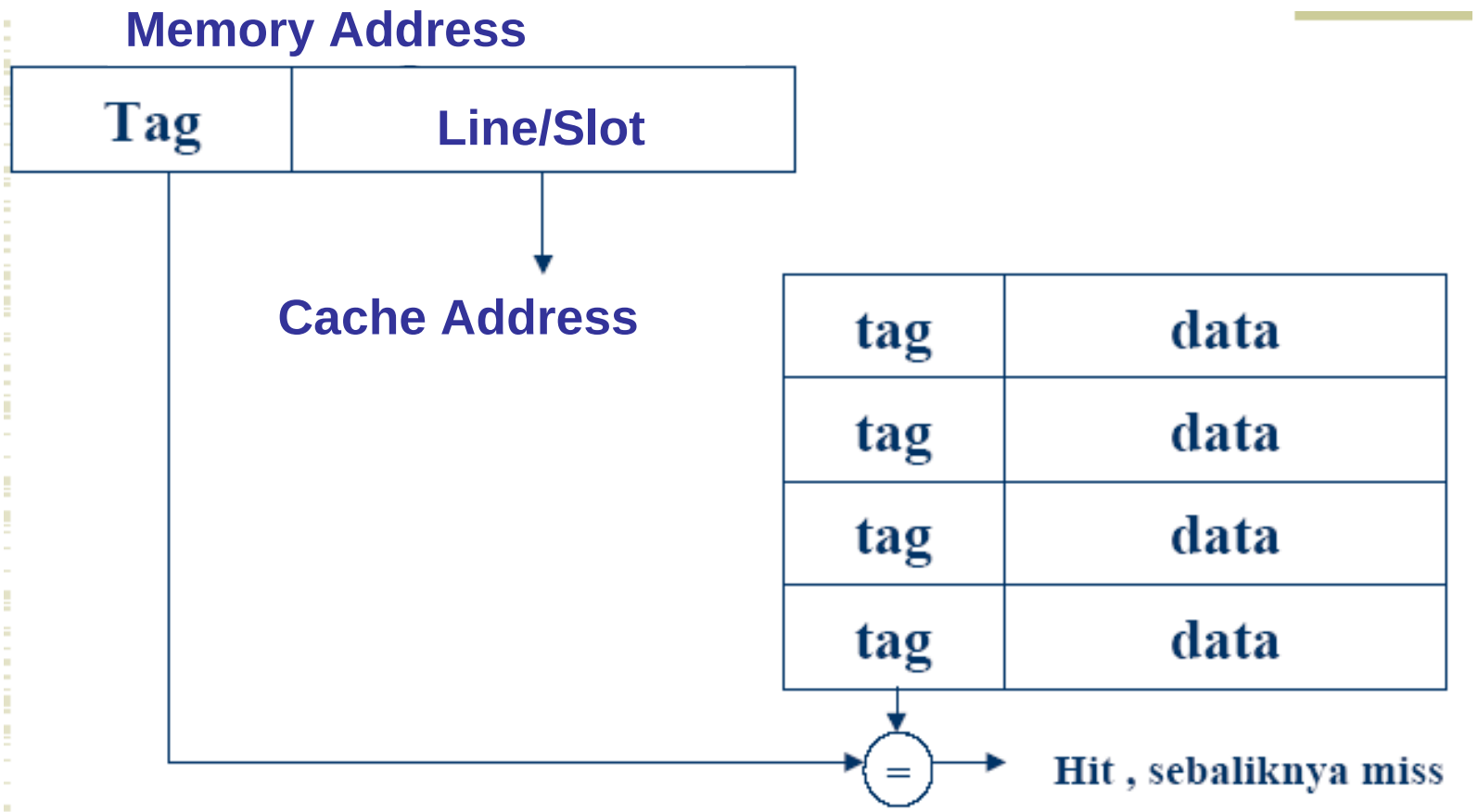
Main memory address = 3 bits

Tag = 1 bit	Line / Slot = 2 bits

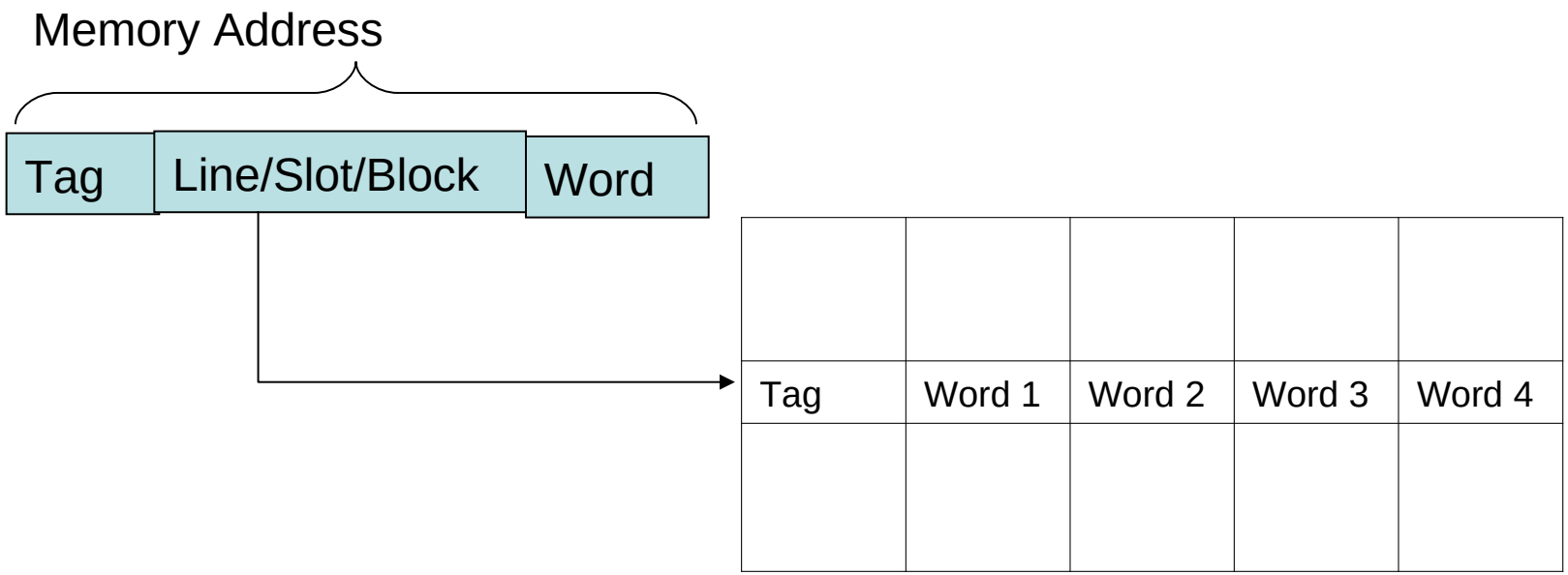
Example 1: Direct Mapping



Direct Mapping: Hit or Miss



Block Direct Mapping

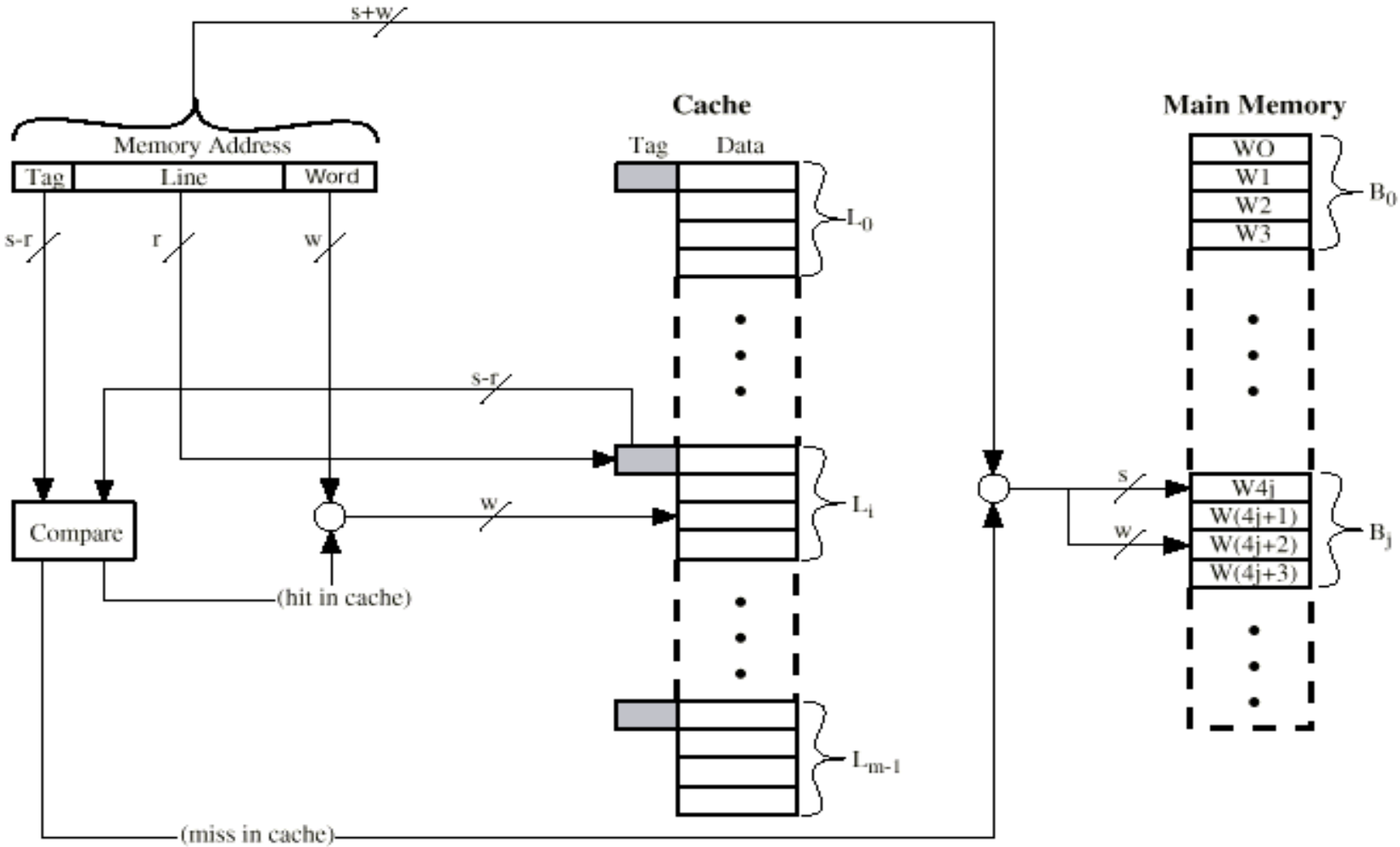


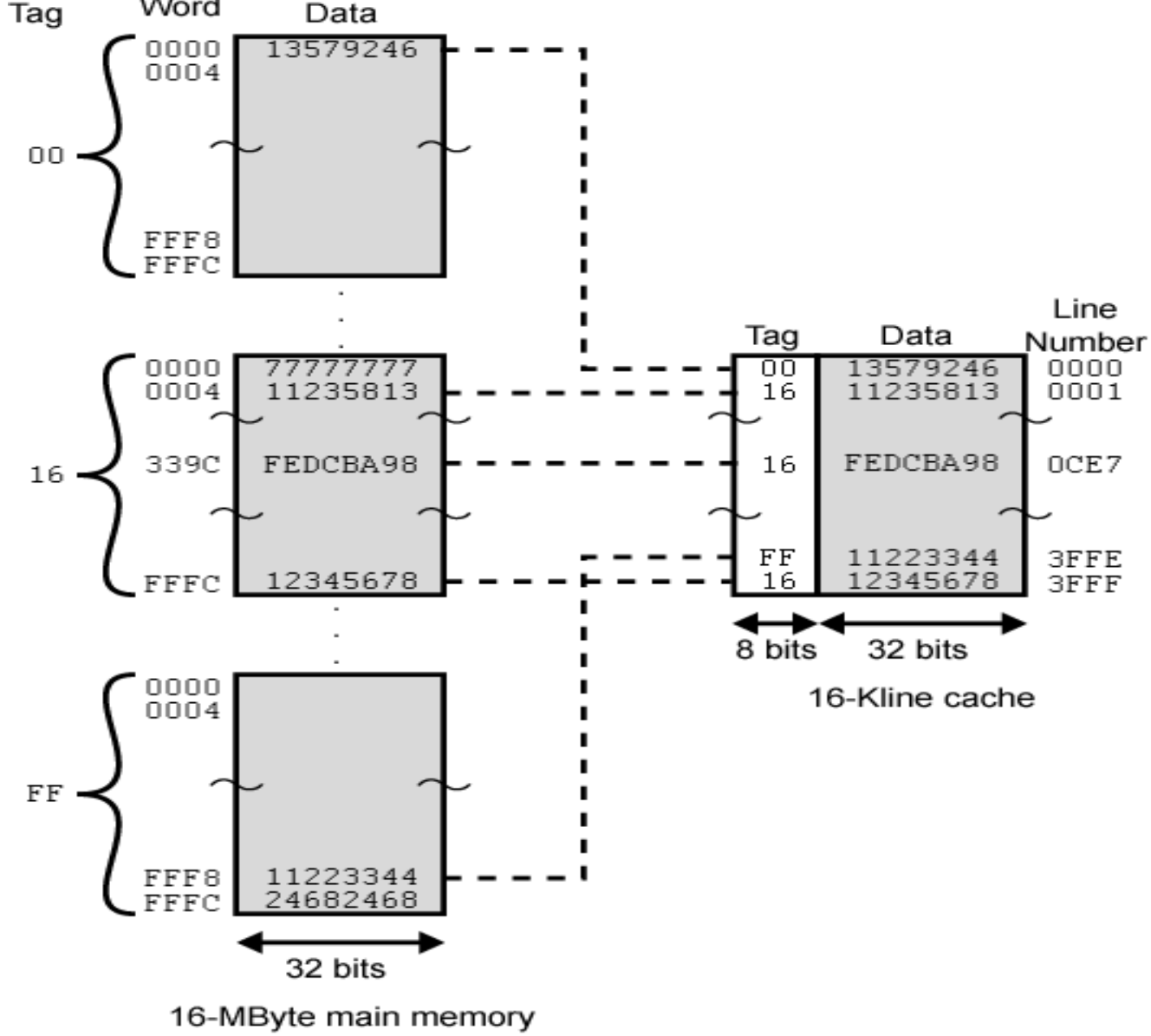
Example: Derive Index(Block) and Offset (Word bits). Given

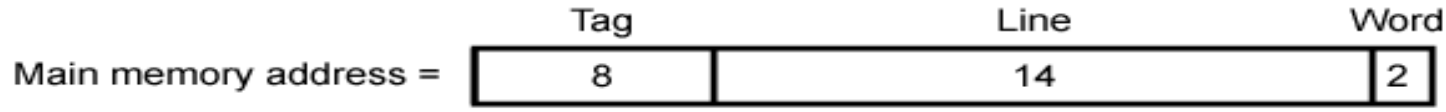
Memory size = 256KB = 2^{18}
Block size = 32Bytes = 2^{5}

Number of blocks in cache = Cache size/Block size = 64KB/32B = $2^{16}/2^{5}$ = 2^{11}

Number of bits in Tag =


s+w
Memory Address
Tag
Line
Word
s-r
r
w
Cache
Tag
Data
L_0
L_i
L_{m-1}
s-r
w
Compare
(hit in cache)
(miss in cache)
Main Memory
WO
W1
W2
W3
B_0
s
w
W4j
W(4j+1)
W(4j+2)
W(4j+3)
B_j


Tag
Word
Data
00
0000
0004
13579246
FFF8
FFFC
16
0000
0004
77777777
11235813
339C
FEDCBA98
FFFC
12345678
FF
0000
0004
FFF8
FFFC
11223344
24682468
32 bits
16-MByte main memory
Tag
Data
Line Number
00
16
13579246
11235813
0000
0001
16
FEDCBA98
0CE7
FF
16
11223344
12345678
3FFE
3FFF
8 bits
32 bits
16-Kline cache

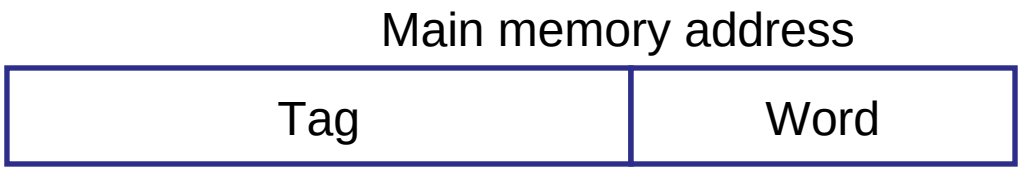

Tag
Line
Word
Main memory address =
8
14
2

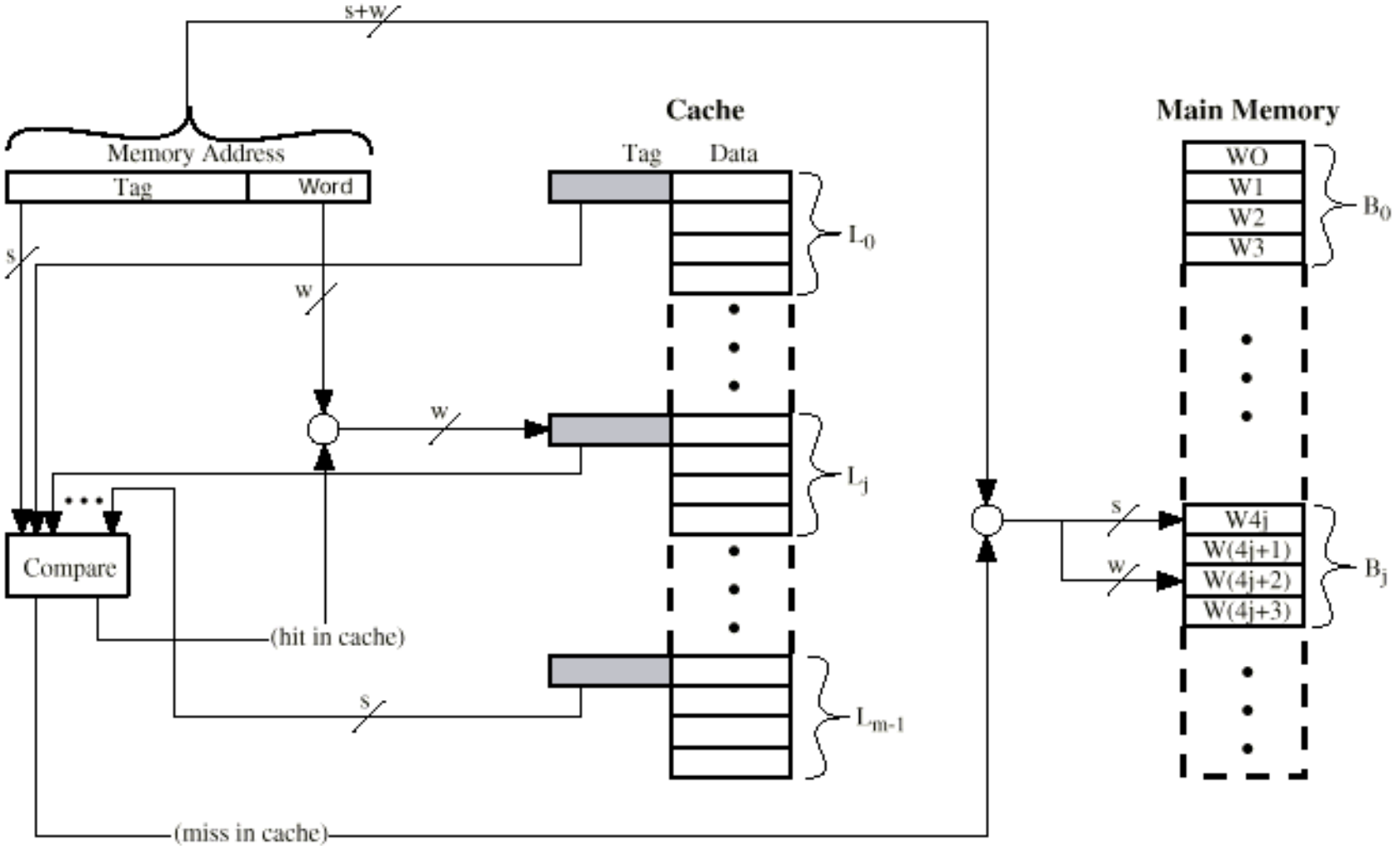
Direct Mapping pros & cons

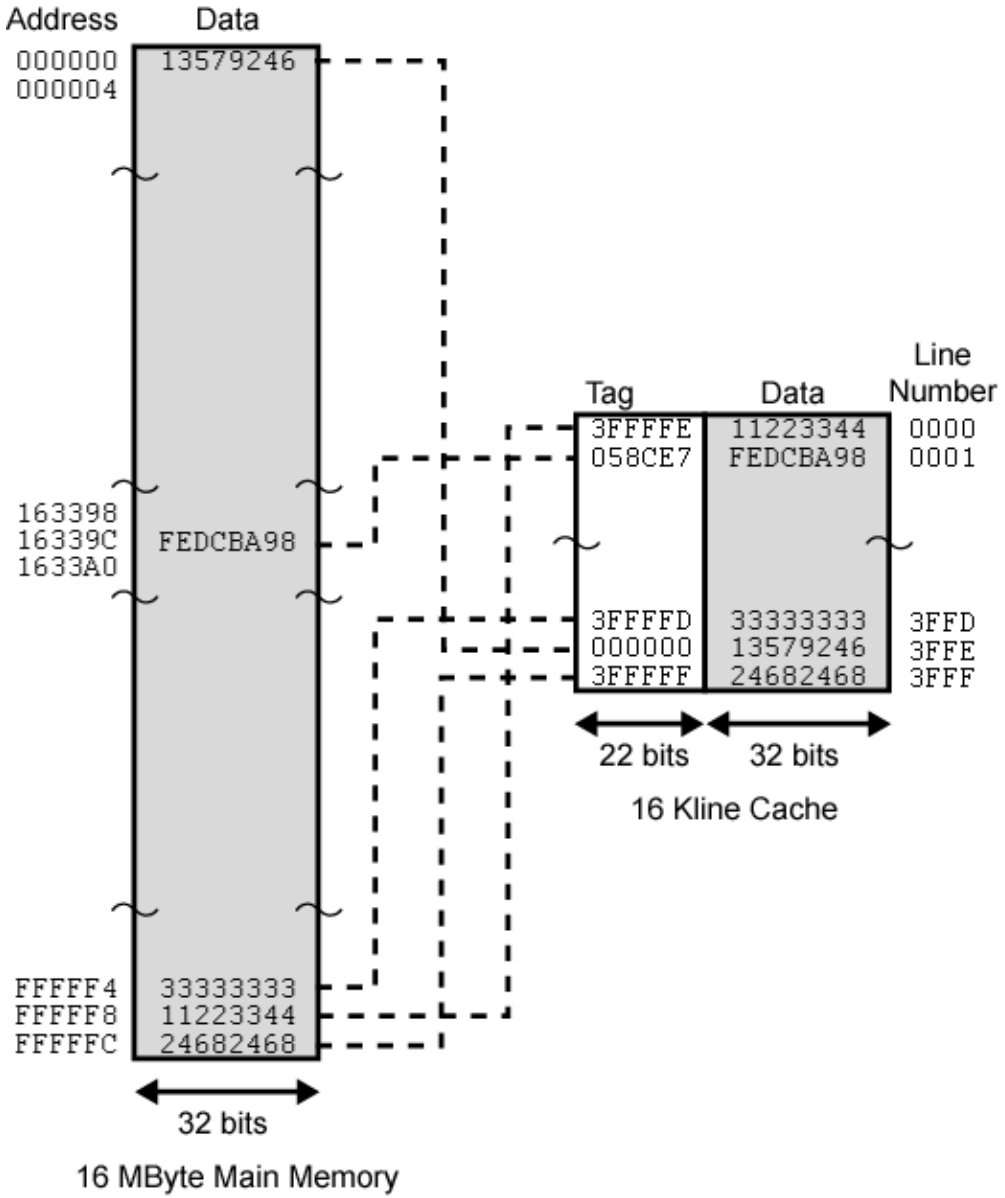
- Simple
- Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

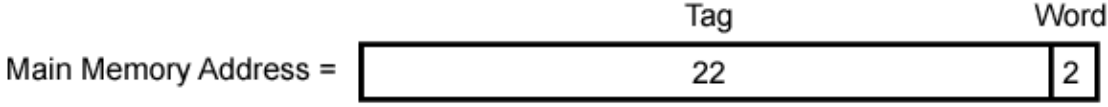
Associative Mapping

- A main memory block can load into any line of cache
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line’s tag is examined for a match
- Cache searching gets expensive




s+w
Memory Address
Tag
Word
s
w
Cache
Tag
Data
L_0
w
L_j
Compare
(hit in cache)
s
L_{m-1}
(miss in cache)
Main Memory
WO
W1
W2
W3
B_0
s
w
W4j
W(4j+1)
W(4j+2)
W(4j+3)
B_j


Address
Data
000000
000004
13579246
163398
16339C
1633A0
FEDCBA98
FFFFF4
FFFFF8
FFFFFC
33333333
11223344
24682468
32 bits
16 MByte Main Memory
Tag
Data
Line Number
3FFFFE
058CE7
11223344
FEDCBA98
0000
0001
3FFFFD
000000
3FFFFF
33333333
13579246
24682468
3FFD
3FFE
3FFF
22 bits
32 bits
16 Kline Cache


Tag
Word
Main Memory Address =
22
2

Example: Given M.M= 16MB, Cache has 16K Lines, Block Size=4words, Devise Memory address format

(a). Main memory capacity = 16MByte =

Therefore the number of bits for the main memory address = 24 bits

(b) Given a cache memory of the size 16 Kline = 2^{14} locations (line number) – not important…! (Since the any word can be loaded into any cache location)

(c) Each line stores a block of words.

Block size =

Therefore 2 bits for block of words

(d) **Memory address format consists of tag and word field.**

Tag bits + words bits = 24 bits

Tag bits + 2 = 24 bits

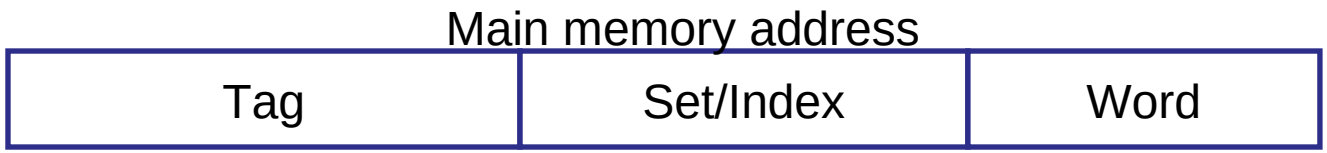
Therefore tag bits = 24 – 2 = 22 bits

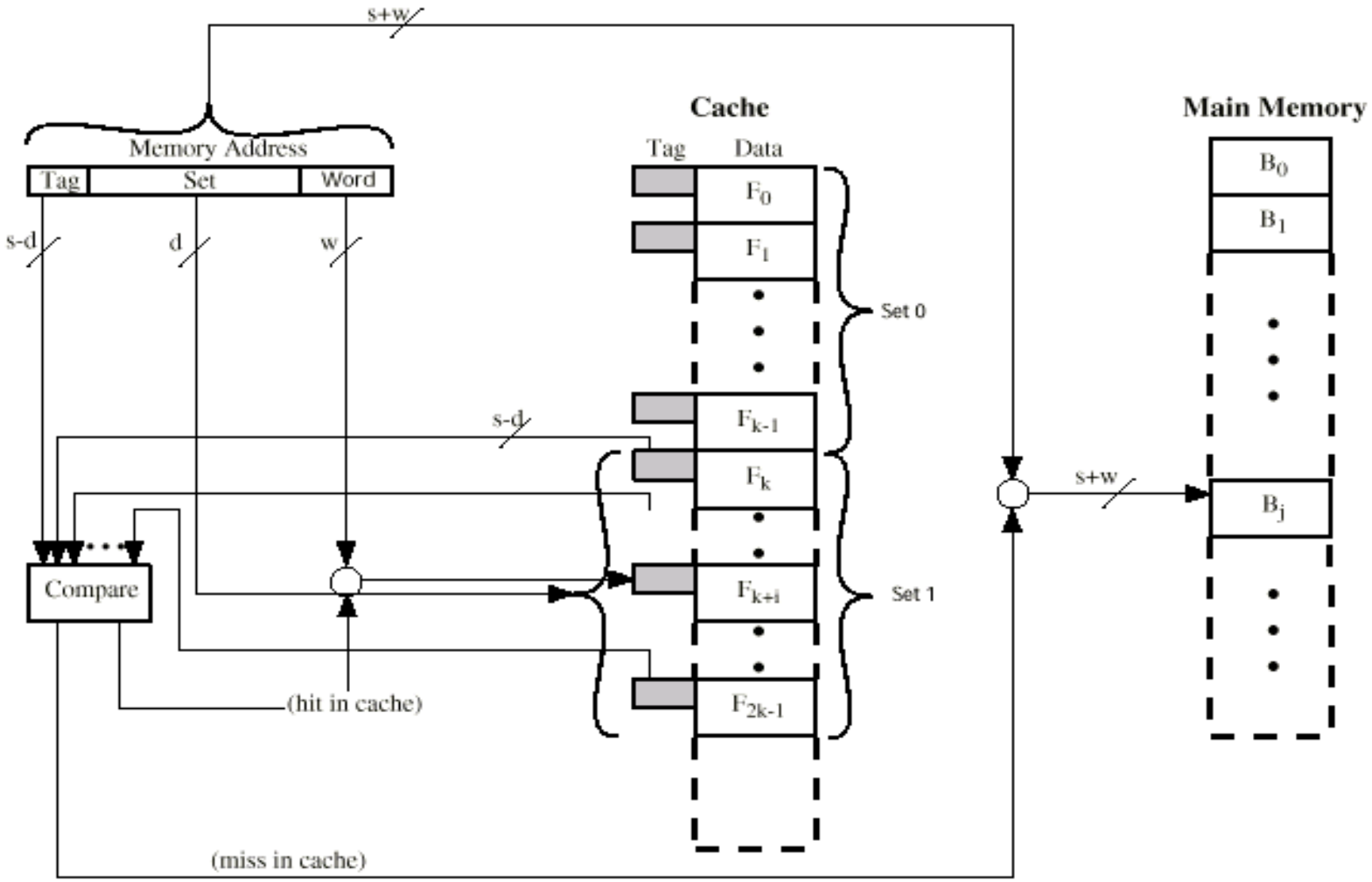
Draw the format of main memory address

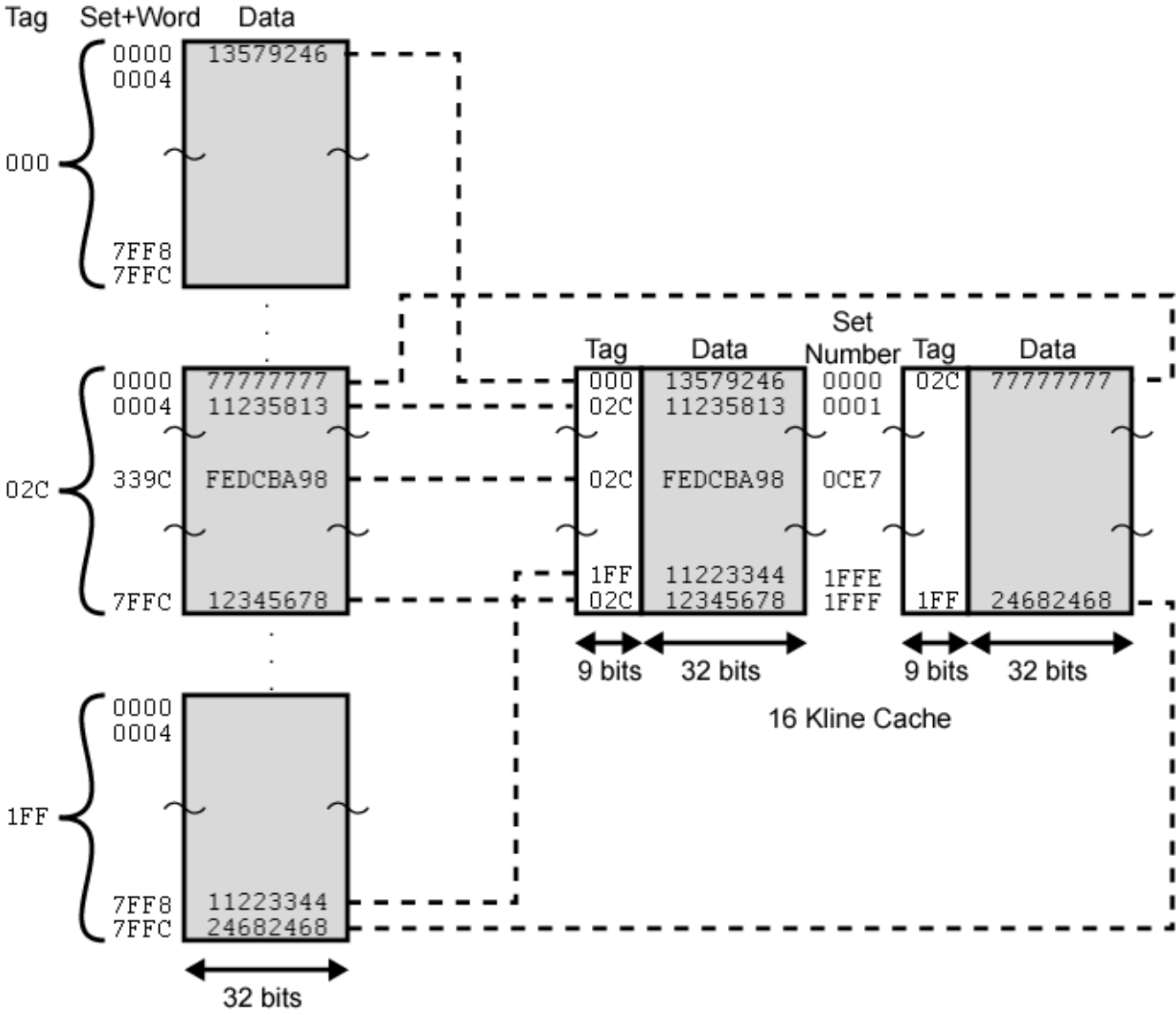
TAG=22bits	Word=2bits

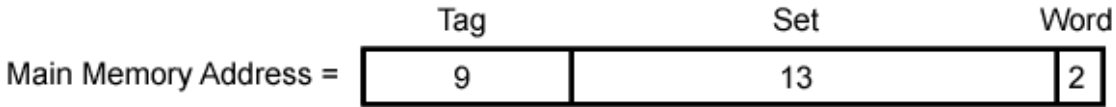
Set Associative Mapping

- Cache is divided into a number of sets
- Each set contains a number of lines
- A given block maps to any line in a given set
  - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - 2 way set associative mapping
  - A given block can be in one of 2 lines in only one set




s+w
Memory Address
Tag
Set
Word
s-d
d
w
Cache
Tag
Data
F_0
F_1
Set 0
F_{k-1}
F_k
F_{k+i}
Set 1
F_{2k-1}
s-d
Compare
(hit in cache)
(miss in cache)
s+w
Main Memory
B_0
B_1
B_j


Tag
Set+Word
Data
0000
0004
13579246
000
7FF8
7FFC
0000
0004
77777777
11235813
02C
339C
FEDCBA98
7FFC
12345678
1FF
0000
0004
7FF8
7FFC
11223344
24682468
32 bits
16 MByte Main Memory
Tag
Data
Set Number
Tag
Data
000
02C
13579246
11235813
0000
0001
02C
02C
FEDCBA98
0CE7
1FF
02C
11223344
12345678
1FFE
1FFF
9 bits
32 bits
16 Kline Cache
02C
1FF
77777777
24682468
9 bits
32 bits


Tag
Set
Word
Main Memory Address =
9
13
2

Set Associative Mapping
Example 1

- Given memory size of 256KB, Cache size of 64KB, and memory block size of 2 Bytes, find the address format using 4-Way Set Associate Mapping ?
- **Solution:**

Memory Size=256KB= 2^{18}

Block Size(Offset) = 2 Bytes = 2^{1}

Cache Size = 64KB=2^{16}

Set Size = 4 blocks

To find:

Number of Sets in Cache(Index) = $2^{16}/(4*2)=2^{13}$

Number of bits in Tag = Total – Index-Offset = 18-13-1 = 4

Tag : 4 bits	Index: 13 bits	Offest : 1 bit

Set Associative Mapping Example 2

- Given memory size of 256KB, Cache size of 64KB, and memory block size of 2 Bytes, find the address format using 8-Way Set Associate Mapping ?
- **Solution:**

Memory Size=256KB= 2^{18}

Block Size(Offset) = 2 Bytes = 2^{1}

Cache Size = 64KB=2^{16}

Set Size = 8 blocks

To find:

Number of Sets in Cache(Index) = $2^{16}/(8*2)=2^{12}$

Number of bits in Tag = Total – Index-Offset = 18-12-1 = 5

Tag : 5 bits	Index: 12 bits	Offest : 1 bit

Replacement Algorithms (1) Direct mapping

- No choice
- Each line/slot or block only maps to one line/slot or block
- Replace that line/slot or block

Replacement Algorithms (2) Associative & Set Associative

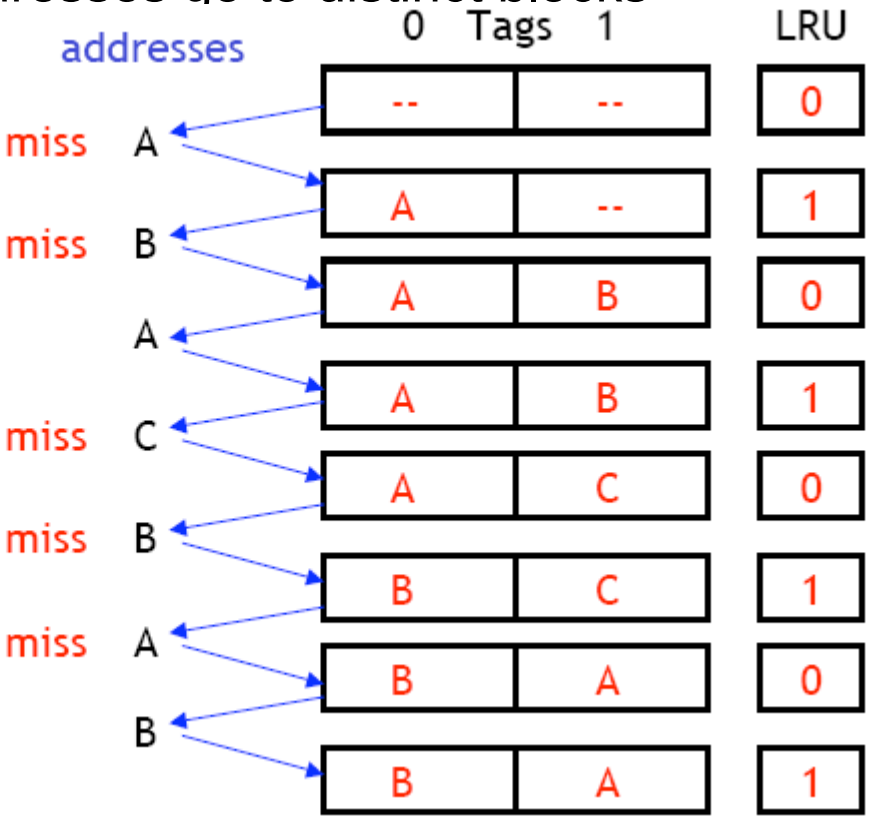
- Hardware implemented algorithm (speed)
- Least Recently used (LRU)
- First in first out (FIFO)
  - replace block that has been in cache longest
- Least frequently used (LFU)
  - replace block which has had fewest hits
- Random

LRU Example

- Assume a associative cache with two blocks, which of the following memory references miss in the cache
  - assume distinct addresses go to distinct blocks

On a miss, we replace the LRU.

On a hit, we just update the LRU.

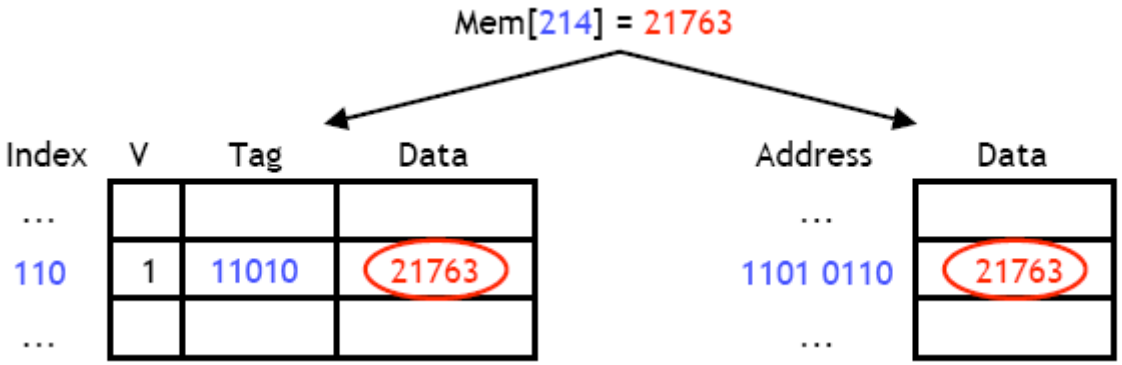


Write Policy

- Must not overwrite a cache block unless main memory is up to date
- Multiple CPUs may have individual caches
- I/O may address main memory directly

Write through

- A write-through cache write updates both the cache *and the main memory*
- 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

Cache Coherence

- Assuming a system with multiple processors and multiple caches.
- If a word in one cache is altered, then that word needs to be altered in :
  - Main memory as well as other cache
- Even with write-through policy other caches may contain invalid data.
- Thus the problem is to maintain Cache Coherence.

Approaches for Cache Coherence

- **Bus watching with write through**
  - Each cache controller monitors the address lines to detect write operations to memory.
  - If shared locations in memory are overwritten by another controller(master).
  - Other cache controllers invalidate their corresponding cache entries.

Approaches for Cache Coherence (Cont'd)

- **Hardware Transparency**
  - All updates to main memory via cache are reflected in all caches.
  - If one of the cache is updated :
    - It is automatically written main memory as well as updating other caches.
- **Noncacheable Memory**
  - Chunks of main memory are designed as non-cacheable.
  - Shared memory is noncacheable.
  - All the access to the shared memory is cache misses.

Line Size

- The relationship between block size and hit ratio is complex.
- As block size increases the hit ratio will increase at first.
  - Locality of reference comes into play
- But very large block size will also decrease performance:
  - Probability of newly fetched information becomes less.
  - Blocks need to be replaced shortly after being brought in!

Line Size (Cont’d)

- Optimal block size is a characteristic of the program.
- Line size of 8 to 64 is optimal for normal usage.
- For HPC systems, line size is 64- to 128 bytes.

Multilevel Caches

- On-chip Cache – L1
- Off-chip Cache – L2
  - Static RAM, faster then system bus
  - Separate data path, reduce burden on system bus
  - A number of CPU have incorporated L2 on the CPU itself.
- Off-chip Cache – L3
  - L2 moves into CPU, L3 becomes external.
  - Most recently, L3 also incorporated on the CPU.

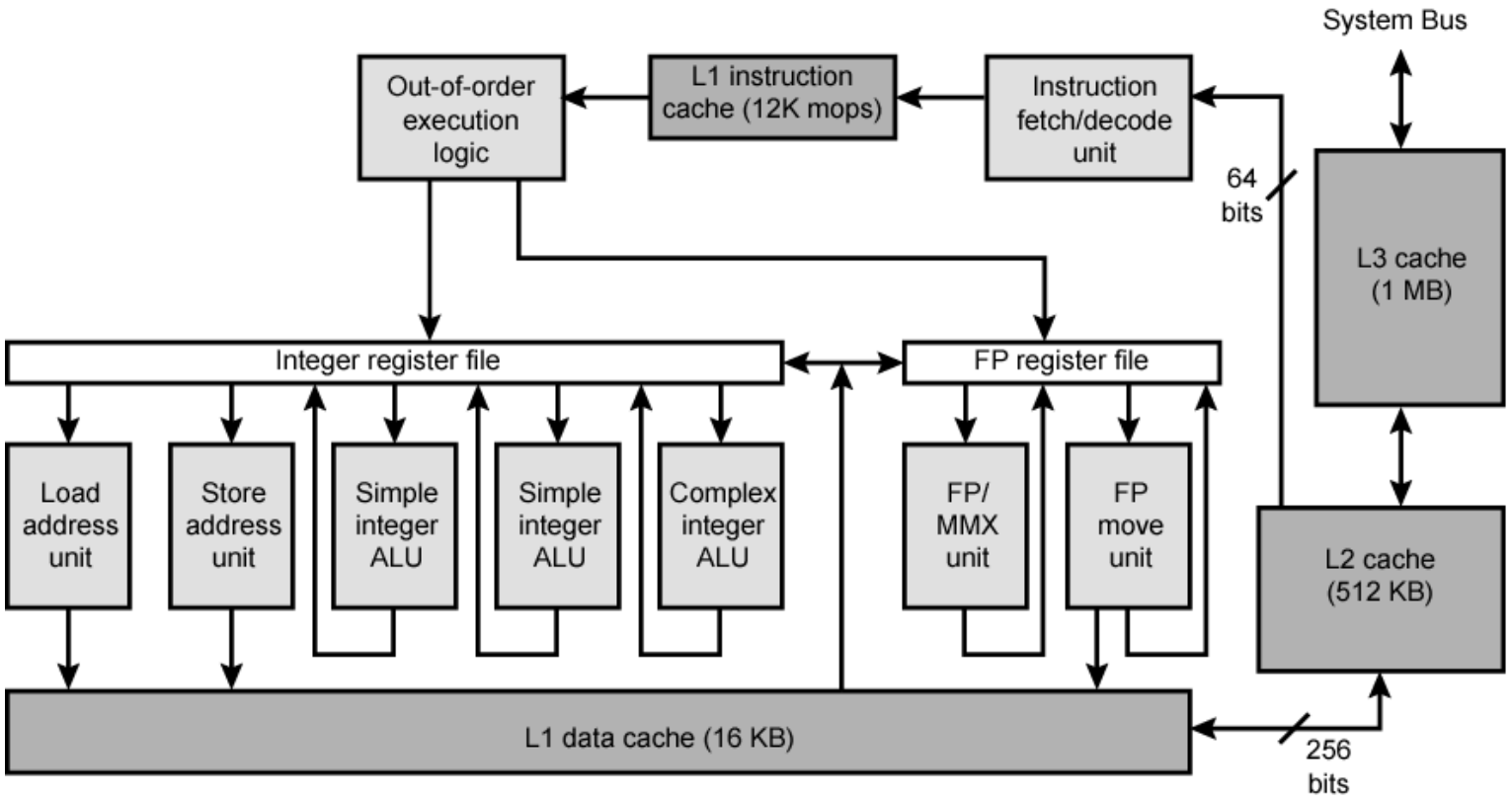
Pentium 4 Cache

- 80386 – no on chip cache
- 80486 – 8k using 16 byte lines and four way set associative organization
- Pentium (all versions) – two on chip L1 caches
  - Data & instructions
- Pentium III – L3 cache added off chip
- Pentium 4
  - L1 caches
    - 8k bytes
    - 64 byte lines
    - four way set associative
  - L2 cache
    - Feeding both L1 caches
    - 256k
    - 128 byte lines
    - 8 way set associative
  - L3 cache on chip

Intel Cache Evolution (Extra Notes-Skip)

Problem	**Solution**	**Processor on which feature first appears**
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip	Add external L2 cache using faster technology than main memory	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Pentium 4 Block Diagram (Extra Notes)



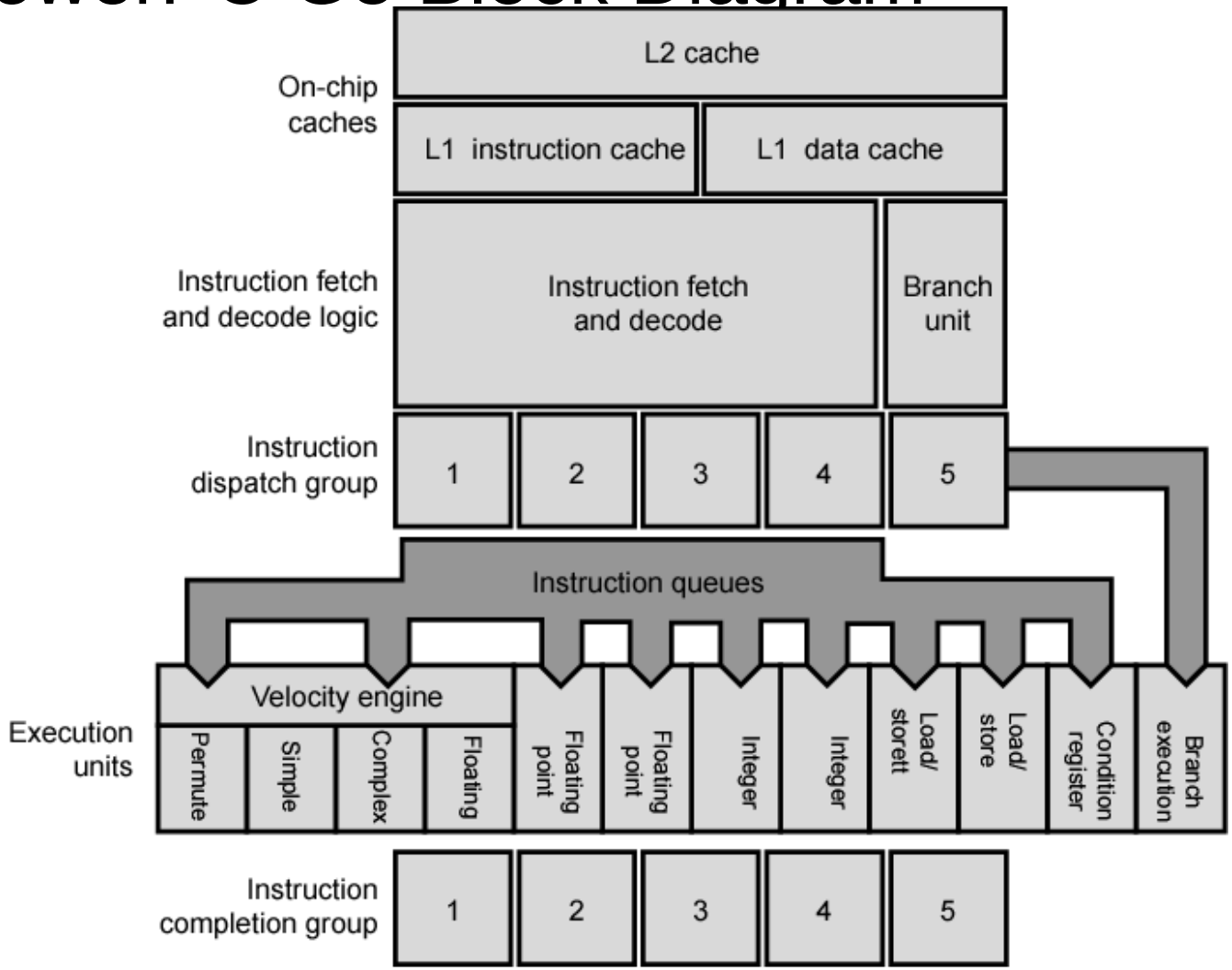
Pentium 4 Core Processor (Extra Notes)

- Fetch/Decode Unit
  - Fetches instructions from L2 cache
  - Decode into micro-ops
  - Store micro-ops in L1 cache
- Out of order execution logic
  - Schedules micro-ops
  - Based on data dependence and resources
  - May speculatively execute
- Execution units
  - Execute micro-ops
  - Data from L1 cache
  - Results in registers
- Memory subsystem
  - L2 cache and systems bus

PowerPC Cache Organization

- 601 – single 32kb 8 way set associative
- 603 – 16kb (2 x 8kb) two way set associative
- 604 – 32kb
- 620 – 64kb
- G3 & G4
  - 64kb L1 cache
    - 8 way set associative
  - 256k, 512k or 1M L2 cache
    - two way set associative
- G5
  - 32kB instruction cache
  - 64kB data cache

PowerPC G5 Block Diagram



Internet Sources

- Manufacturer sites
  - Intel
  - IBM/Motorola
- Search on cache

Memory System Performance

- The hit time is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles
- The miss penalty is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least)
- The miss rate is the percentage of misses

Average Memory Access Time

- The average memory access time, or AMAT, can then be computed

  AMAT = Hit time + (Miss rate x Miss penalty)

- This is just averaging the amount of time for cache hits and the amount of time for cache misses
- Obviously, a lower AMAT is better
- Miss penalties are usually much greater than hit times, so the best way to lower AMAT is to reduce the miss penalty *or the miss rate*

Performance Example 1

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

$$\begin{aligned} \text{AMAT} &= \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty}) \\ &= 1 \text{ cycle} + (3\% \times 20 \text{ cycles}) \\ &= 1.6 \text{ cycles} \end{aligned}$$

- If the cache was perfect and never missed, the AMAT would be one cycle. But even with just a 3% miss rate, the AMAT here increases 1.6 times!

Peformance Example 2

A CPU has access to 2 levels of memory. Level 1 contains 1000 words and has access time 0.01 μs; level 2 contains 100,000 words and has access time 0.1 μs. Assume that if a word to be accessed is in level 1, then, the CPU accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the CPU. For simplicity, we ignored the time required for the CPU to determine whether the word is in level 1 or level 2.

Suppose 95% of the memory access are found in the cache. Then, the average access to a word can be expressed as:

Peformance Example 3

(0.95)(0.01 μs) + (0.05)(0.01 μs +0.1 μs)

= 0.0095 + 0.0055

= 0.015 μs.