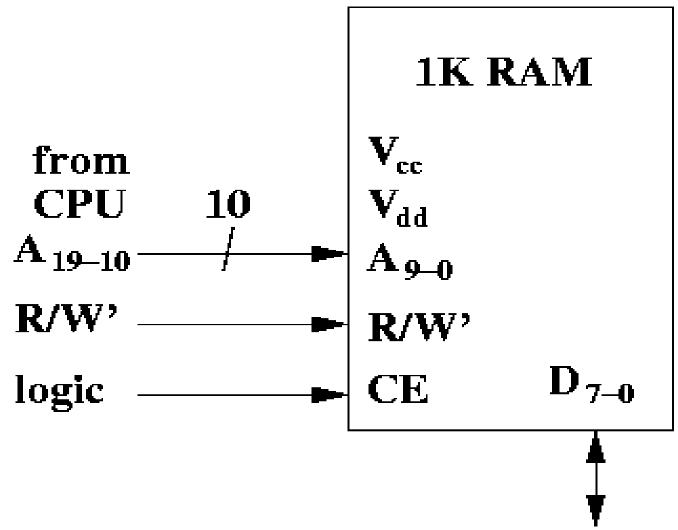
Memory

What do we use for accessing small amounts of data quickly? Registers (32 in MIPS)
Why not store all data and instructions in registers?
Too much overhead for addressing; lose speed advantage
Register file can use 5-32 decoder or 32-1 MUX to select
Memory (RAM) is organized in larger quantities



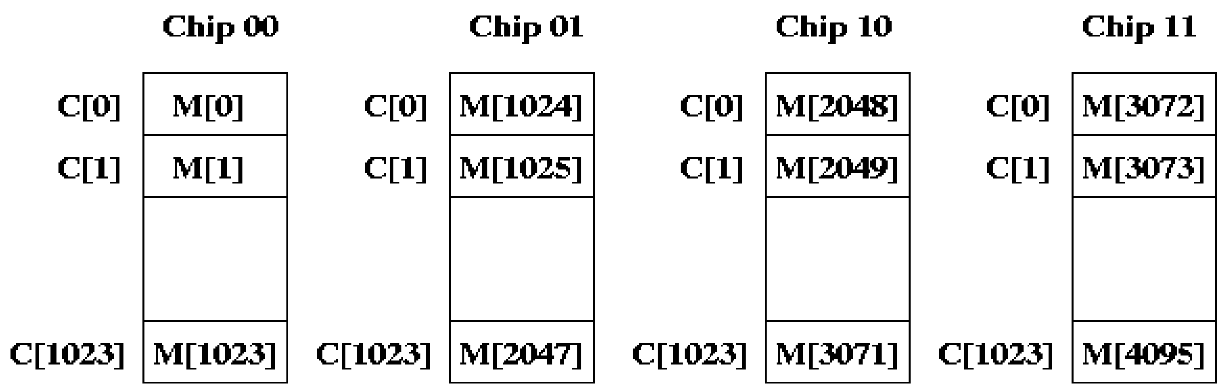
V_{cc}, V_{dd}: voltages (5 volts and ground), necessary to power the chip, but do not affect logic
A_{9-0}: Address input (10 bits) Why 10 bits?
R/W': read/not write, selects read or write
1: read, 0: write
CE: chip enable, allows read or write; when 0, neither read nor write
also called chip select
D_{7-0}: 8 bits of data read or to be written
may be bidirectional, or 16 separate lines (pins on the chip)
Where do the inputs come from? CPU

Memory

How do we get larger amounts of memory?

Think of memory as abstraction

- **4K memory: like byte array M[4095]**
- **Give memory index of byte, get data value back**
- **Use 4 1K chips**



Each chip has elements C[0] up to C[1023]

Chips are numbered 00, 01, 10, 11

Each chip contains 1024 of the elements (0 to 1023, 1024 to 2047, etc.)

Where is element M[1025]?

- **Chip 01 at index 1**

Where is element M[3071]?

- **Chip 10 at index 1023**

Memory

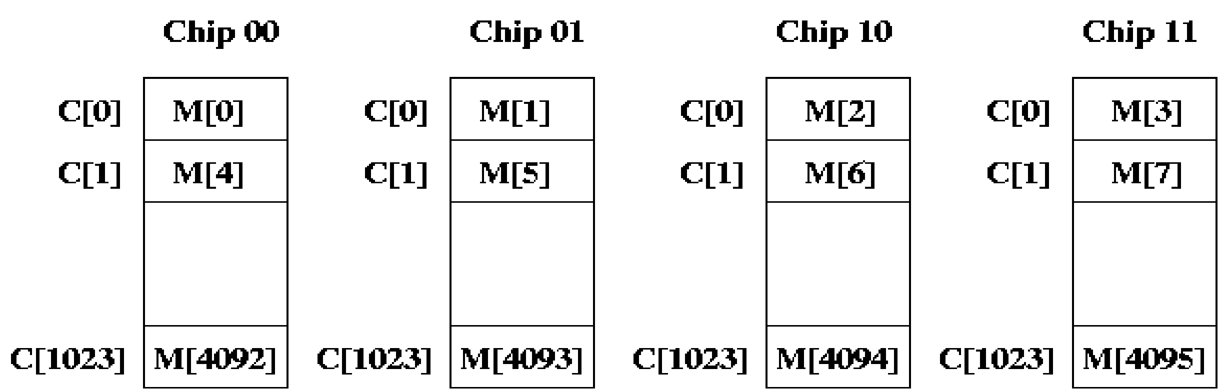
Problem:

How long does it take to get 4 bytes?

If the data is word-aligned, and it takes time T to get 1 byte, it will take time 4T to get 4 bytes (4 accesses to the same chip)

Why is word alignment significant?

Solution: put consecutive bytes on different chips. This is called interleaving.



M[0] on Chip 00, M[1] on Chip 01, M[2] on Chip 10, M[3] on Chip 11, and M[4] back on Chip 00

If data is word-aligned, first byte always appears on chip 00

Also, the other bytes are at the same address on the other chips

Allows accessing an entire word in time T

Where is M[5]? Chip 01, index 1

In general, where is M[i]? Think of the address in binary.

The chip is at i % 4 — **The chip is the low two bits.**

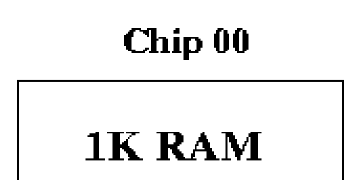
The index is at i / 4 — **The index is bits $B_{11\text{-}2}$.**

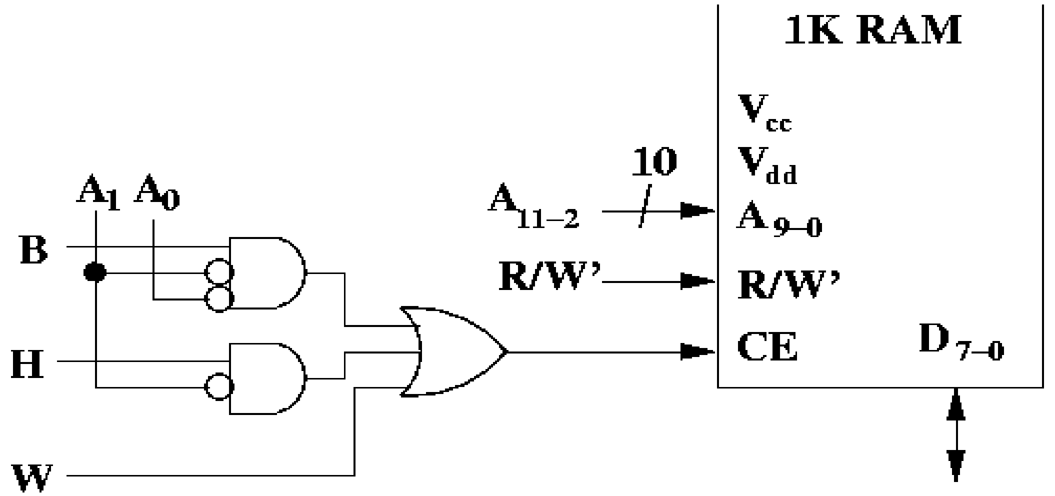
Memory: chip enable

Need to generate signals to determine which addresses to access in memory

Logic for chip enable

- **CPU can generate 3 control signals:**
  - **B which indicates that the CPU wants to access a byte**
  - **H which indicates that the CPU wants to access a halfword**
  - **W which indicates that the CPU wants to access a word**
- **Which chips are enabled?**
  - **B: 1 chip**
  - **H: 2 chips**
  - **W: all 4 chips**
- **Address patterns:**
  - **addresses on chip 00 end in 00 (divisible by 4)**
  - **addresses on chip 01 end in 01 (congruent to 1 mod 4)**
  - **addresses on chip 10 end in 10 (congruent to 2 mod 4)**
  - **addresses on chip 11 end in 11 (congruent to 3 mod 4)**
- **Logic for chip 00:**
  - **If W = 1, then all four chips are enabled**
  - **If B = 1, select chip 00 when $A_1A_0 = 00$**
  - **If H = 1, select Chip 00 if $A_1 = 0$ (already know that $A_0 = 0$)**
- **Boolean expression:**
  - $CE = W + (H * A_1') + (B * A_1'A_0')$





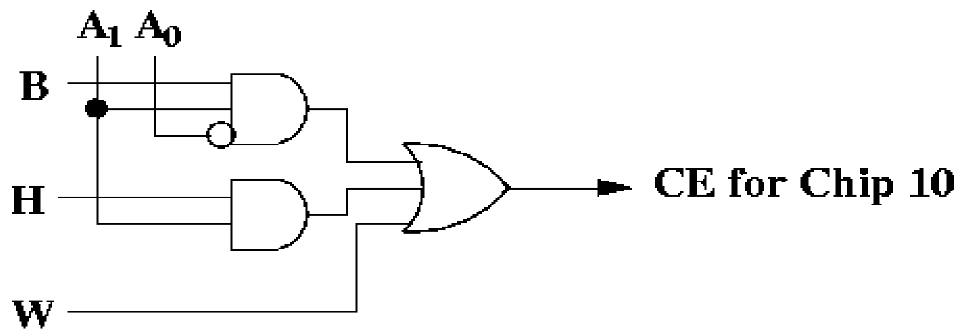
Notice address bits $A_{9\text{-}0}$:

To get M[i], access index i / 4

Same as shifting to the right by 2 bits, or accessing bits 11-2 from the address bus

Chip 10:

$CE = W + (H * A_1) + (B * A_1 A_0')$



Invalid addresses:

we assume that the other 20 bits of the address are all 0

Memory: locality

Memory hierarchy

Registers: small and fast

RAM: large and slow

Ideal: large and fast

Solution: range of memories from fast to slow

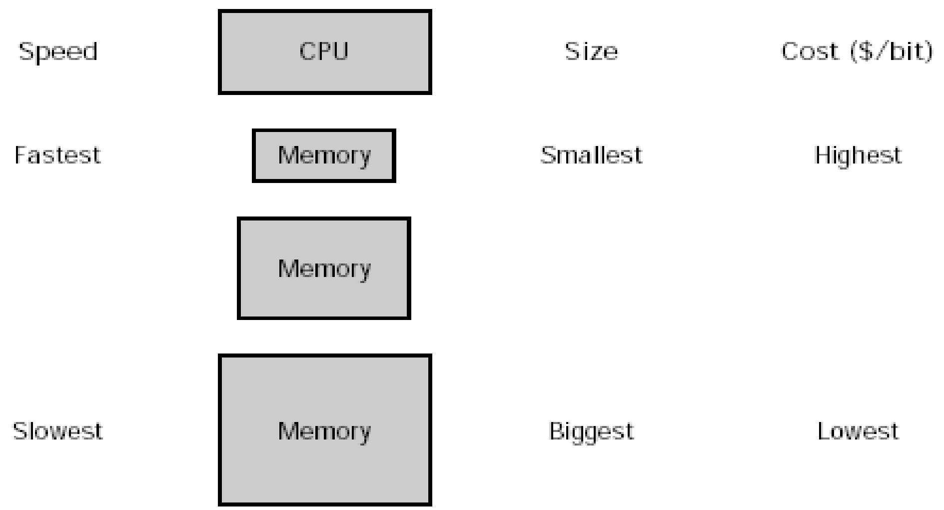


Fig 7.1

If memory accesses are random, can't do much, but programs typically have:

spatial locality: successive memory accesses tend to be close together in location

- **sequential execution**
- **branches tend to be relatively small**
- **arrays**

temporal locality: location, once accessed, will tend to be accessed again within a small amount of time (loop: instructions and data)

Memory: cache

Memory hierarchy

As distance from CPU increases, so does size

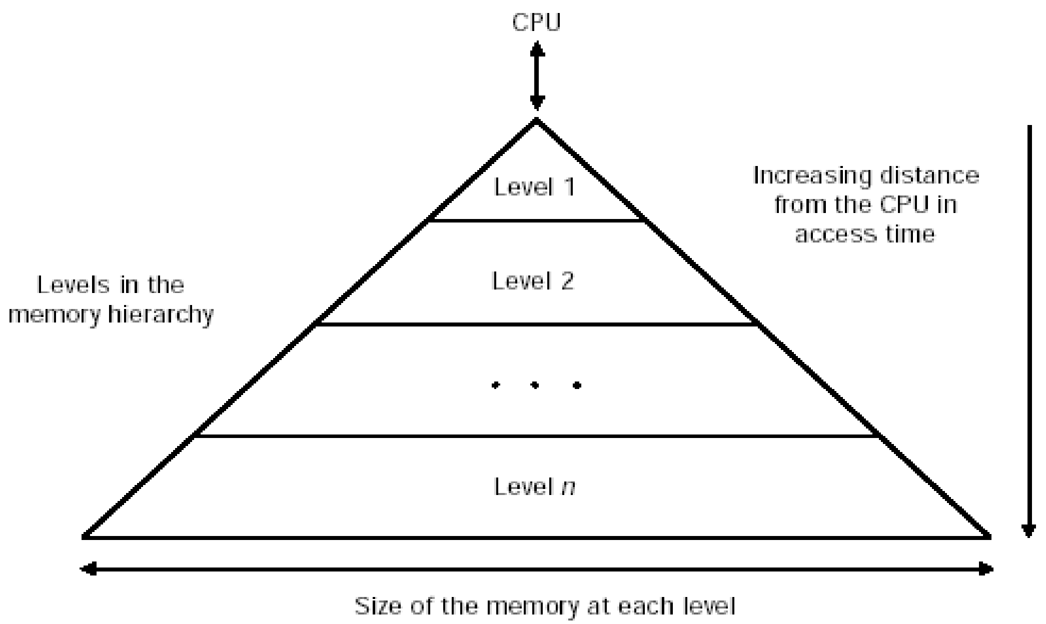


Fig 7.3

To take advantage of:

temporal locality: keep recently used data closer to CPU

spatial locality: when moving data to a higher level, move a contiguous block

Miss: requested data not found in currrent level

Hit: requested data is found in current level

Hit rate: how often a requested data item is found in a given level of the hierarchy

If hit rate at the top levels of the hierarchy is large, then the average access time will be close to the fastest access time

Cache: level in hierarchy between CPU and main memory

Memory: levels

Data is transferred only between adjacent levels:

When miss occurs at one level of hierarchy, data is transferred from next lower level

Minimum unit of data transferred: block

Performance depends on speed of hits and misses

Hit time: time to access upper level,

including determining hit or miss

Miss penalty: time to access lower level to get data

Issues

How much data to transfer between levels

Policy to replace data in upper levels

Policy to update data in each level

Analogy

Need 10 books for a term paper

Bring all 10 books back to your desk,

instead of going back to library 10 times

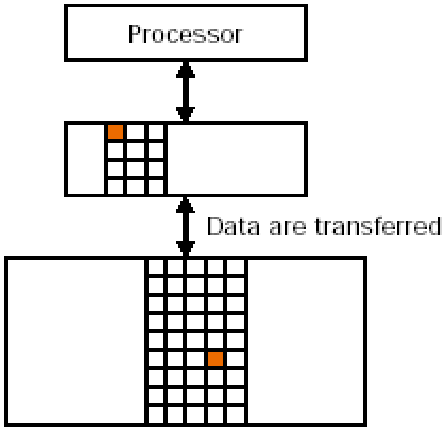
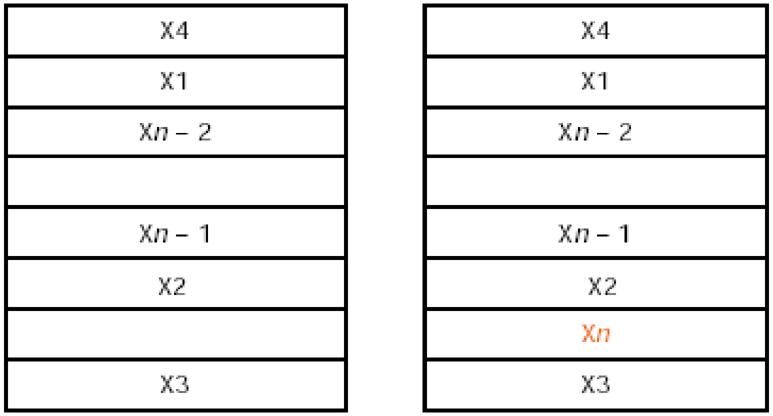


Fig. 7.2

Memory: cache



a. Before the reference to Xn   b. After the reference to Xn

Fig 7.4

Consider simple cache

Processor requests are each 1 word

Cache blocks are 1 word

Processor makes reference to word Xn, and Xn is not currently in the cache: cache miss

If there is space in the cache, then word Xn is copied into the cache

2 questions:

How do we know if a data item is in the cache now?

If so, where is it located?

Simplest method:

Base cache address directly on memory address: direct mapped cache

(Notice they are not equal, since the cache is smaller)

Typical method:

(block address) mod (number of cache blocks in whole cache)

Easy to compute if number of cache blocks is power of 2: use low order n bits of address

where n is $\log_2$ (number of cache blocks)

Example:

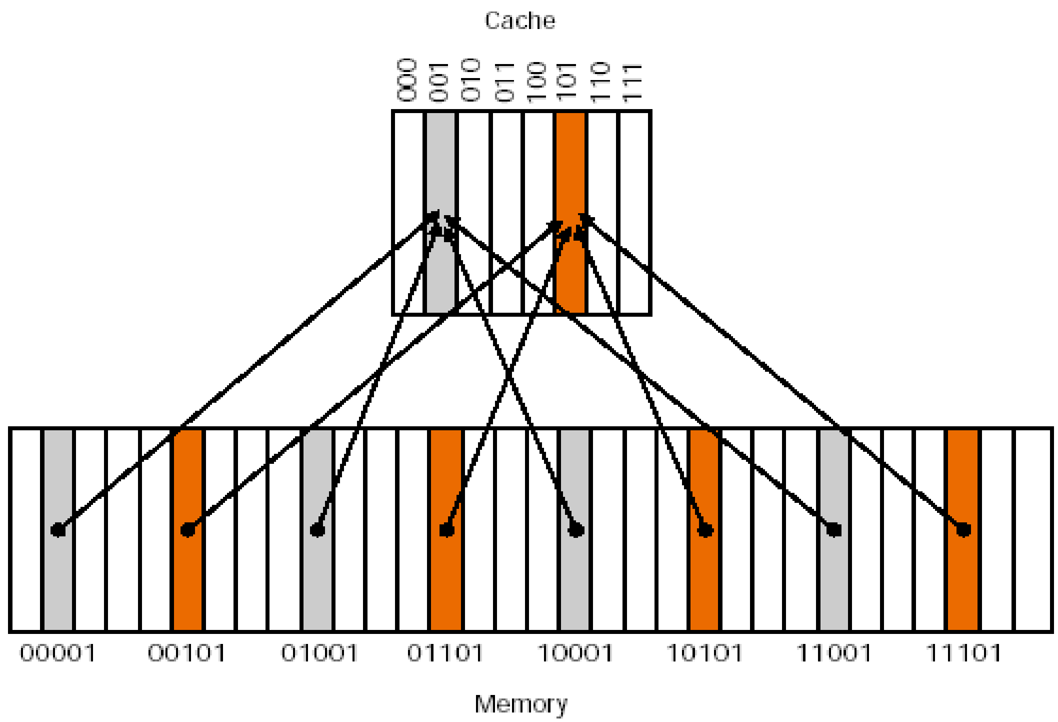


Fig 7.5

Cache of 8 words

Each data word is mapped to location whose address ends in same 3 bits

For example, all the gray words have addresses ending in 001,

mapped to cache block 001

Each cache location can contain several possible data words

If a word is in the cache, how do we know which one it is?

Add tags to the cache entries

Tag needs to contain only the upper bits of the address

In example, only need upper 2 bits for tag

Also need to recognize whether block is empty: valid bit

Managing cache

How to manage cache efficiently?

Temporal locality

copy data into cache when accessed

Spatial locality

copy 2^k block of data including accessed data item

How to choose range of addresses to copy?

Might choose data from addr - delta to addr + delta

where delta is 2^{k-1}

However, this is not so convenient to manage

Instead, use all addresses with same upper n-k bits

Example:

Want to access data with address A_{31-0}.

Copy 32 bytes with addresses:

A_{31-5} **00000**

A_{31-5} **00001**

A_{31-5} **00010**

...

A_{31-5} **11111**

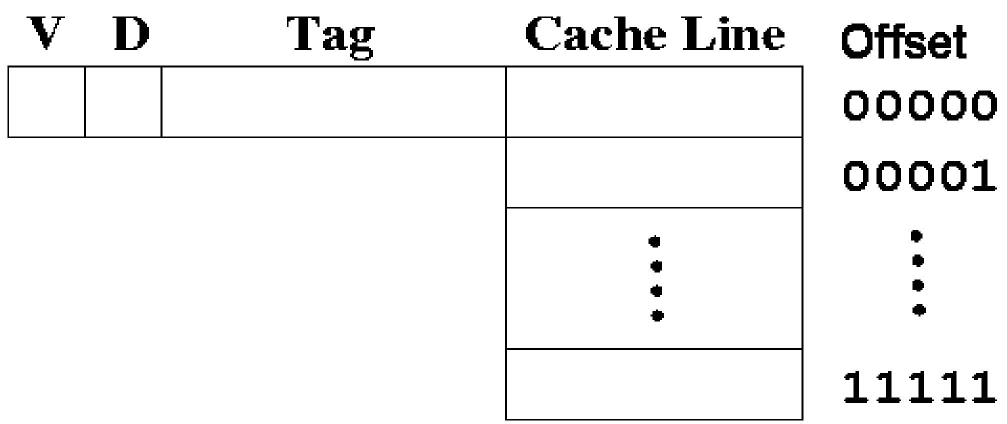
These 32 bytes are called a cache line.

The upper 27 bits are the tag.

Managing cache

Cache line is stored in cache slot:

- Actual data: cache line (data block)
  - Offset: k-bit address of data within cache line
- V: valid bit
- D: dirty bit
  - If data block valid, D = 1 indicates data has been modified since put in cache
- Tag: upper 32-k bits



Single Slot of Cache

Number of bits per cache slot in this example:

2 bits for V and D + 27 bits for tag + 32 * 8 bits for data = 285

Managing cache: issues

Cache misses: types

- compulsory: first reference to a data item
- capacity: not enough space in cache
  - too few slots
  - cache line too small
- conflict: space is available, but data block already stored at that location
  - 2 cache lines map to same cache slot

Instruction and data cache

- instructions and data have different access patterns, use different memory areas
- use separate instruction and data caches
  - one reason to have separate instruction and data memories in datapath

Modifying data in the cache

- write-back: update main memory only when block is removed from cache
  - saves time required to write main memory for each store
- write-through
  - update main memory at the same time as cache
  - save time by continuing execution while main memory write completes

Replacement policy

- how to choose cache line to replace
  - LRU: least recently used: slot which has not been used in the longest time
  - LFU: least frequently used
  - FIFO: first in, first out: slot which has been in the cache the longest
  - Random: may be only 10% worse than LRU
- may require additional hardware to keep track

Cache: fully-associative

More flexible cache management

- fully-associative

Assume cache consists of $2^7 = 128$ slots, with $2^5 = 32$ bytes per cache line

Address A_{31-0} consists of tag bits A_{31-5} and offset A_{4-0}

31 … 5	4 … 0
Tag	Offset

If data not in the cache, pick a slot

- Fully-associative cache: may go in any slot
  - Pick one with V = 0
  - If none, evict a slot using replacement policy (LRU, FIFO, etc.)

How do we know if a cache line is in the cache?

- Must search every slot, but hardware can search in parallel, unlike software
- Compare the address tag bits with the tags of each slot in the cache
  - Can be done using a comparator (combinational circuit using XNOR gates)
- Must also have valid bit V = 1

Complexity of hardware to manage fully-associative cache slows down the speed
of the cache, so it is not generally used

Cache: set-associative

Hybrid between direct-mapped and fully-associative cache: set-associative

Again assume 128 slots, 32 bytes per slot

Group slots into sets of 8 each (16 sets)

Use lg 16 = 4 bits to specify set

31 9	8 5	4 0
Tag	Set	Offset

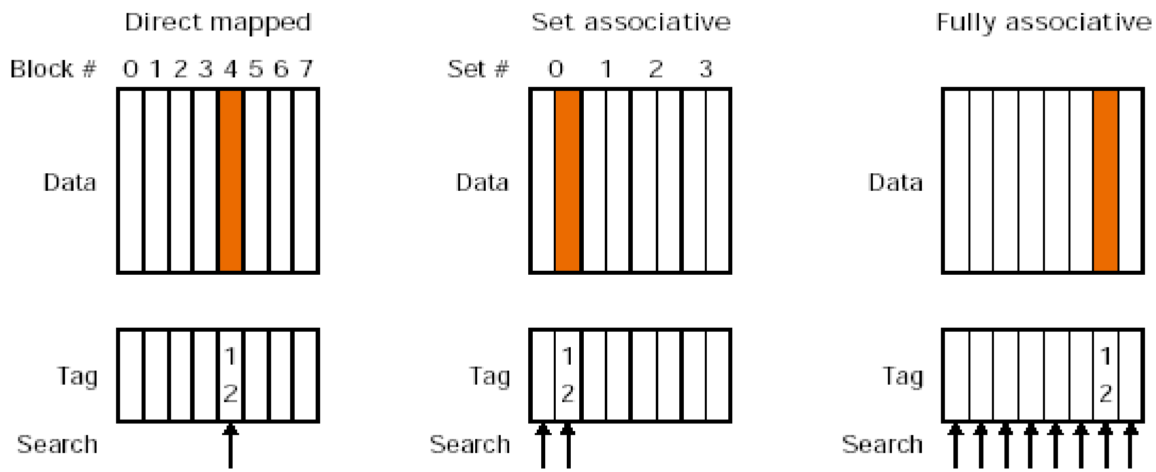
How to find data address B_{31-0}?

- **Use bits B_{8-5} to find the set**
- **There are 8 slots with addresses**
  - B_{8-5}000
  - B_{8-5}001
  - B_{8-5}010
  - ...
  - B_{8-5}111
- **Search all 8 slots to see if the tag B_{31-9} matches the tag of the slot**
- **If so, get the data byte at offset B_{4-0} in the slot**
- **If not, find a slot to use (possibly by eviction), and store 32 bytes in the slot**
- **This example is 8-way set-associative cache**
- **In general, we can have N-way set-associative cache**

Compromise: have to search fewer number of slots, simpler comparison hardware

- **But still have the flexibility of N cache lines per slot (fewer collisions)**

Cache: associativity



Direct-mapped: location determined directly by block number **Fig 7.15**

12 % 8 = 4

Set-associative (2-way): search 2 tags

12 % 4 = 0

Fully-associative: search all tags

Cache: associativity

Range of associativity for 8-block cache

direct-mapped

- **1-way**

set-associative

- **2-way**
- **4-way**

fully-associative

- **8-way**

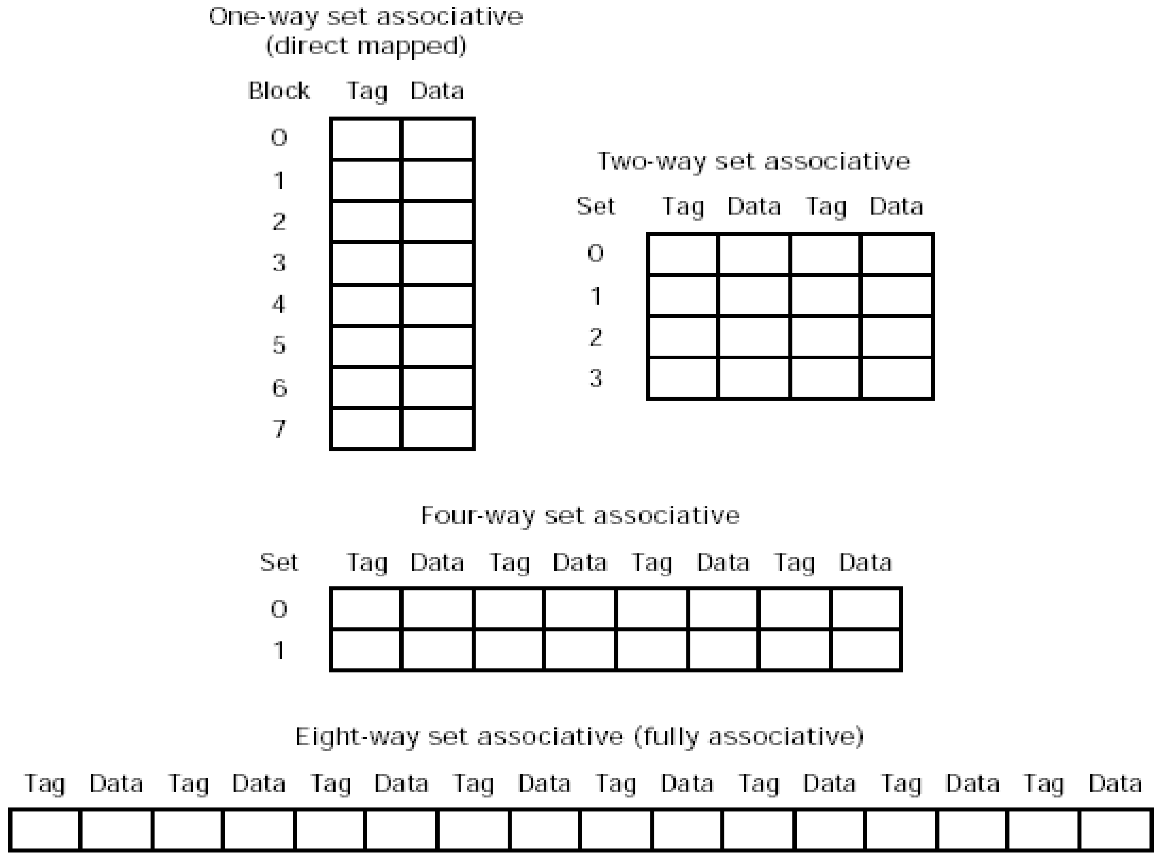


Fig 7.16

Cache: other uses

- Other uses of cache principle
  - Multiple levels
    - Pentium III
      - L1 cache: 16K
      - L2 cache: 512K
  - Disk cache: keep file data blocks in memory
  - Web cache: keep web pages on:
    - local hard drive
    - local server
  - Virtual memory
    - Extend memory hierarchy beyond RAM to disk
    - How big is 32-bit address space (number of valid addresses)?
    - Do most systems have that much memory?
    - Do programmers want to use as much memory as possible?
    - Solution: use RAM as a cache for data blocks on disk
    - Advantages
      - Single program can use very large address space
      - Multiple programs can share same physical memory
      - Memory access can be protected between programs