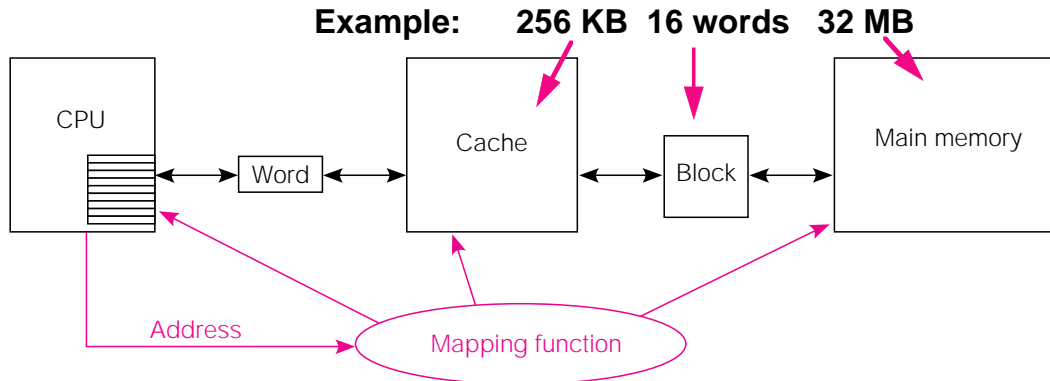


## Fig 7.30 The Cache Mapping Function



The cache mapping function is responsible for all cache operations:

- **Placement strategy**: where to place an incoming block in the cache
- **Replacement strategy**: which block to replace upon a miss
- **Read and write policy**: how to handle reads and writes upon cache misses

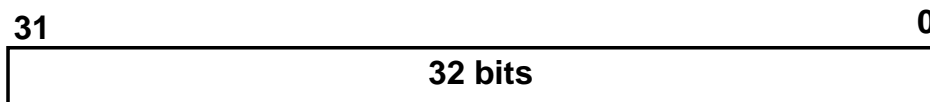
Mapping function must be implemented in hardware. (Why?)

Three different types of mapping functions:

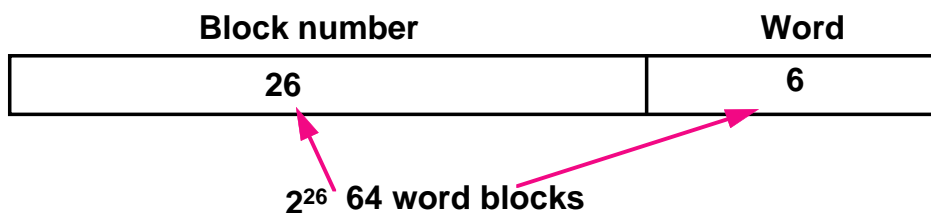
- **Associative**
- **Direct mapped**
- **Block-set associative**

## Memory Fields and Address Translation

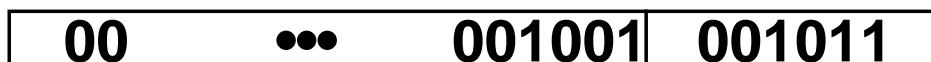
Example of processor-issued 32-bit virtual address:



That same 32-bit address partitioned into two fields, a **block** field, and a **word** field. The word field represents the offset into the block specified in the block field:

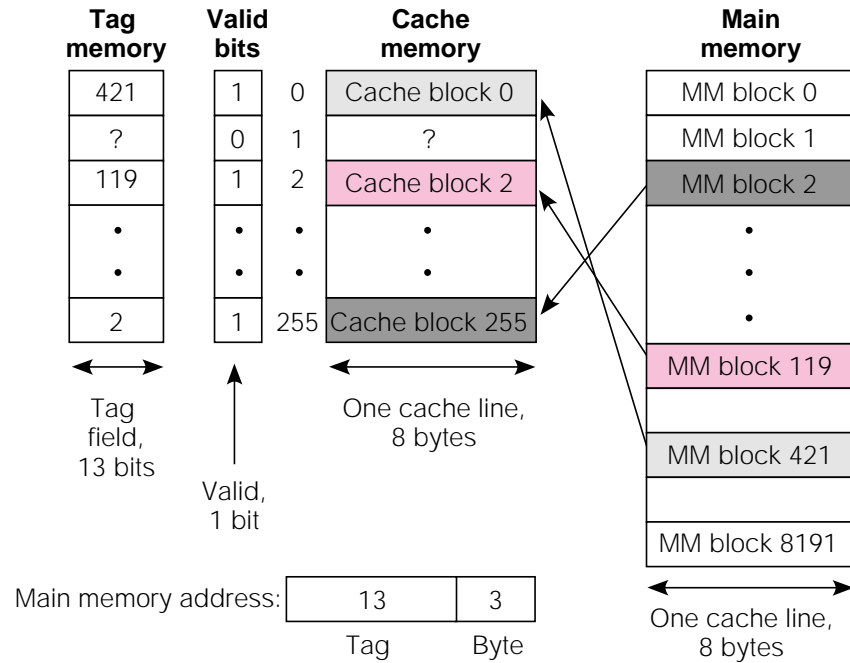


Example of a specific memory reference: Block 9, word 11.



# Fig 7.31 Associative Cache

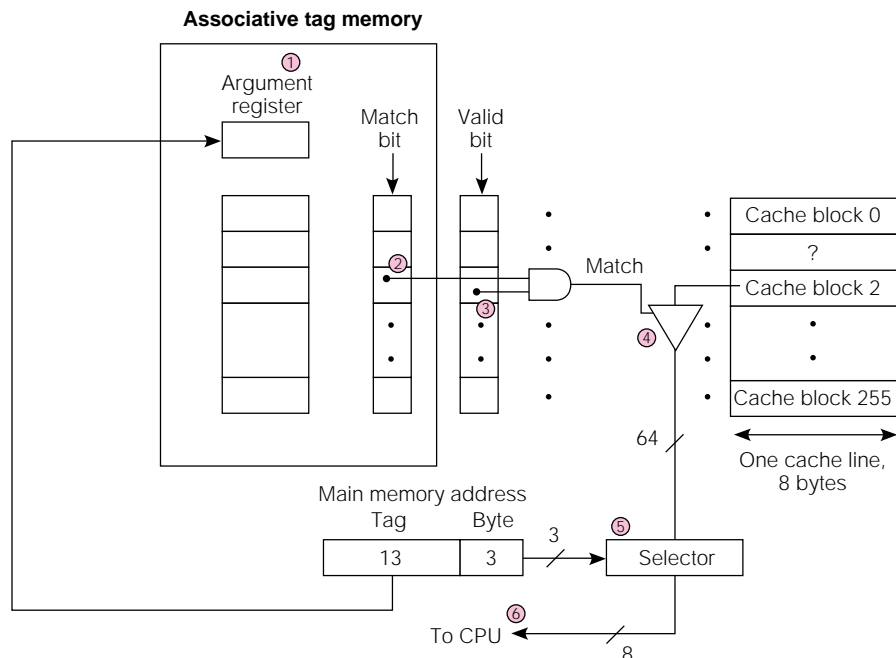
**Associative mapped cache model: any block from main memory can be put anywhere in the cache. Assume a 16-bit main memory.\***



**\*16 bits, while unrealistically small, simplifies the examples**

# Fig 7.32 Associative Cache Mechanism

**Because any block can reside anywhere in the cache, an *associative* (content addressable) memory is used. All locations are searched simultaneously.**



# Advantages and Disadvantages of the Associative Mapped Cache

## Advantage

- Most flexible of all—any MM block can go anywhere in the cache.

## Disadvantages

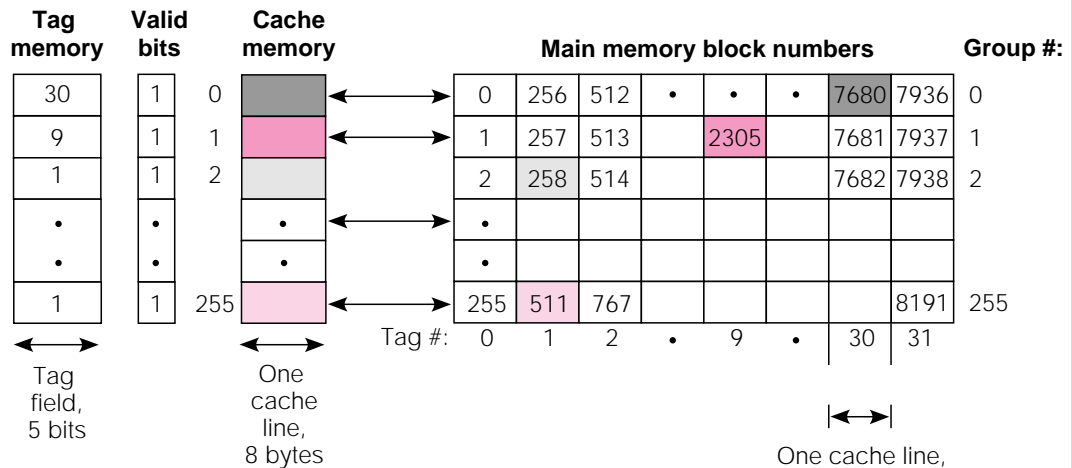
- Large tag memory.
- Need to search entire tag memory simultaneously means lots of hardware.

Replacement Policy is an issue when the cache is full. **—more later—**

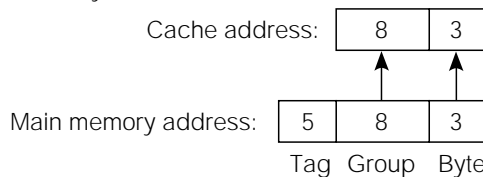
Q.: How is an associative search conducted at the logic gate level?

Direct-mapped caches simplify the hardware by allowing each MM block to go into only one place in the cache:

## Fig 7.33 Direct-Mapped Cache



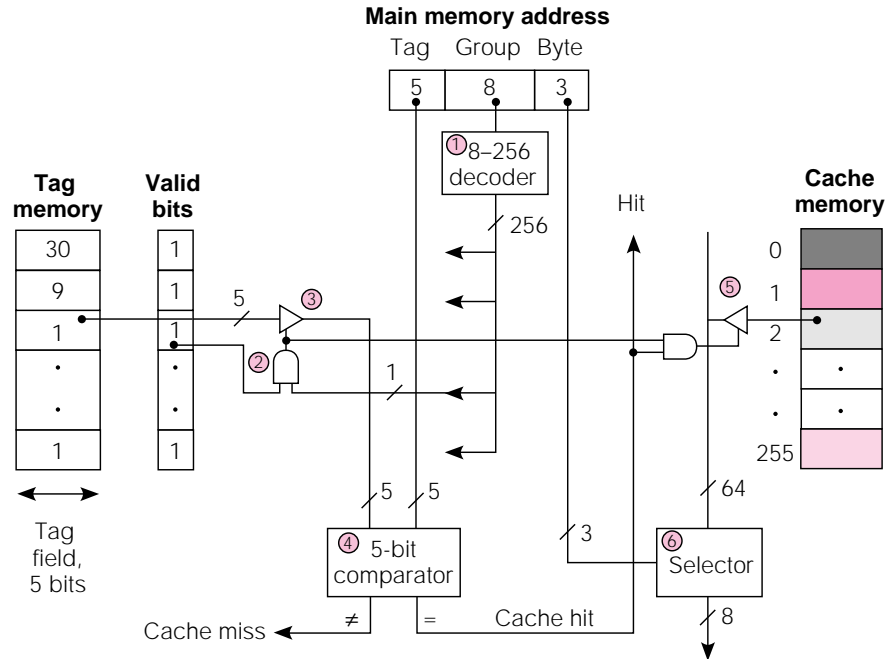
**Key Idea:** all the MM blocks from a given group can go into only one location in the cache, corresponding to the group number.



Now the cache needs only examine the single group that its reference specifies.

## Fig 7.34 Direct-Mapped Cache Operation

1. Decode the group number of the incoming MM address to select the group
2. If Match AND Valid
3. Then gate out the tag field
4. Compare cache tag with incoming tag
5. If a hit, then gate out the cache line

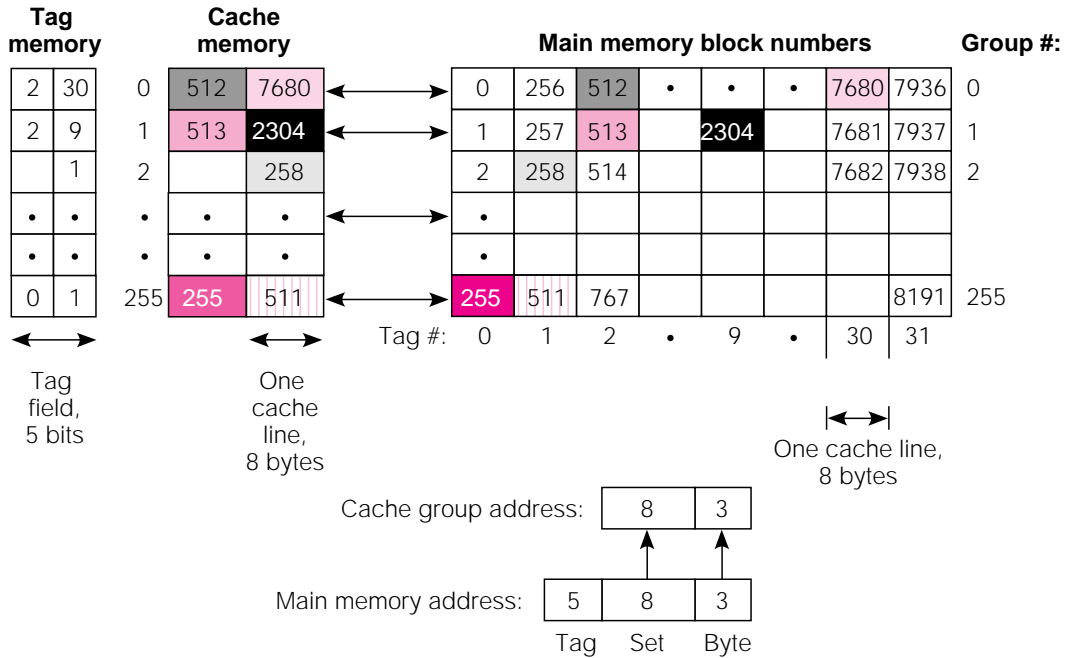


## Direct-Mapped Caches

- The direct mapped cache uses less hardware, but is much more restrictive in block placement.
- If two blocks from the same group are frequently referenced, then the cache will “thrash.” That is, repeatedly bring the two competing blocks into and out of the cache. This will cause a performance degradation.
- Block replacement strategy is trivial.
- Compromise—allow several cache blocks in each group—the **Block-Set-Associative Cache**:

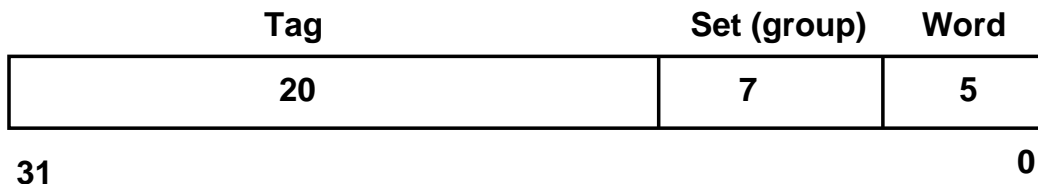
# Fig 7.35 2-Way Set-Associative Cache

Example shows 256 groups, a set of two per group.  
 Sometimes referred to as a **2-way set-associative cache**.



## Getting Specific: The Intel Pentium Cache

- The Pentium actually has two separate caches—one for instructions and one for data. Pentium issues 32-bit MM addresses.
  - Each cache is 2-way set-associative
  - Each cache is 8 K =  $2^{13}$  bytes in size
  - $32 = 2^5$  bytes per line.
  - Thus there are 64 or  $2^6$  bytes per set, and therefore  $2^{13}/2^6 = 2^7 = 128$  groups
  - This leaves  $32 - 5 - 7 = 20$  bits for the tag field:



This “cache arithmetic” is important, and deserves your mastery.

## Cache Read and Write Policies

- Read and Write cache **hit** policies
  - **Writethrough**—updates both cache and MM upon each write.
  - **Write back**—updates only cache. Updates MM only upon block removal.
    - “**Dirty bit**” is set upon first write to indicate block must be written back.
- Read and Write cache **miss** policies
  - Read miss—bring block in from MM
    - Either forward desired word as it is brought in, **or**
    - Wait until entire line is filled, then repeat the cache request.
  - Write miss
    - **Write-allocate**—bring block into cache, then update
    - **Write-no-allocate**—write word to MM without bringing block into cache.

## Block Replacement Strategies

- Not needed with direct-mapped cache
- Least Recently Used (LRU)
  - Track usage with a counter. Each time a block is accessed:
    - Clear counter of accessed block
    - Increment counters with values less than the one accessed
    - All others remain unchanged
  - When set is full, remove line with highest count
- Random replacement—replace block at random
  - Even random replacement is a fairly effective strategy

# Cache Performance

Recall **Access time**,  $t_a = h \cdot t_p + (1 - h) \cdot t_s$  for primary and secondary levels.

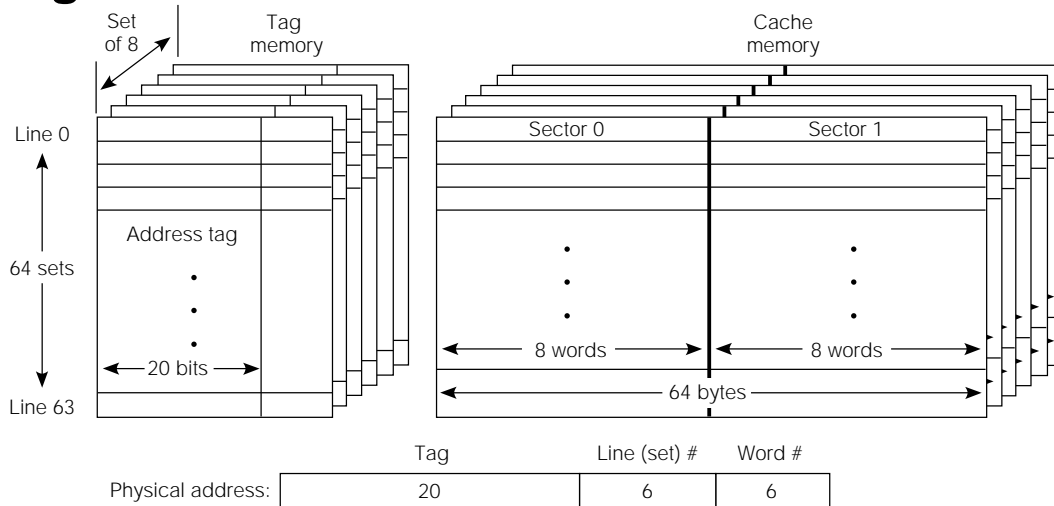
For  $t_p = \text{cache}$  and  $t_s = \text{MM}$ ,

$$t_a = h \cdot t_c + (1 - h) \cdot t_M$$

We define **S**, the **speedup**, as  $S = T_{\text{without}}/T_{\text{with}}$  for a given process, where  $T_{\text{without}}$  is the time taken without the improvement, cache in this case, and  $T_{\text{with}}$  is the time the process takes with the improvement.

Having a model for cache and MM access times and cache line fill time, the speedup can be calculated once the hit ratio is known.

## Fig 7.36 The PowerPC 601 Cache Structure



- The PPC 601 has a *unified* cache—that is, a single cache for both instructions and data.
- It is 32 KB in size, organized as 64 x 8 block-set associative, with blocks being 8 8-byte words organized as 2 independent 4-word sectors for convenience in the updating process
- A cache line can be updated in two single-cycle operations of 4 words each.
- Normal operation is write back, but write through can be selected on a per line basis via software. The cache can also be disabled via software.