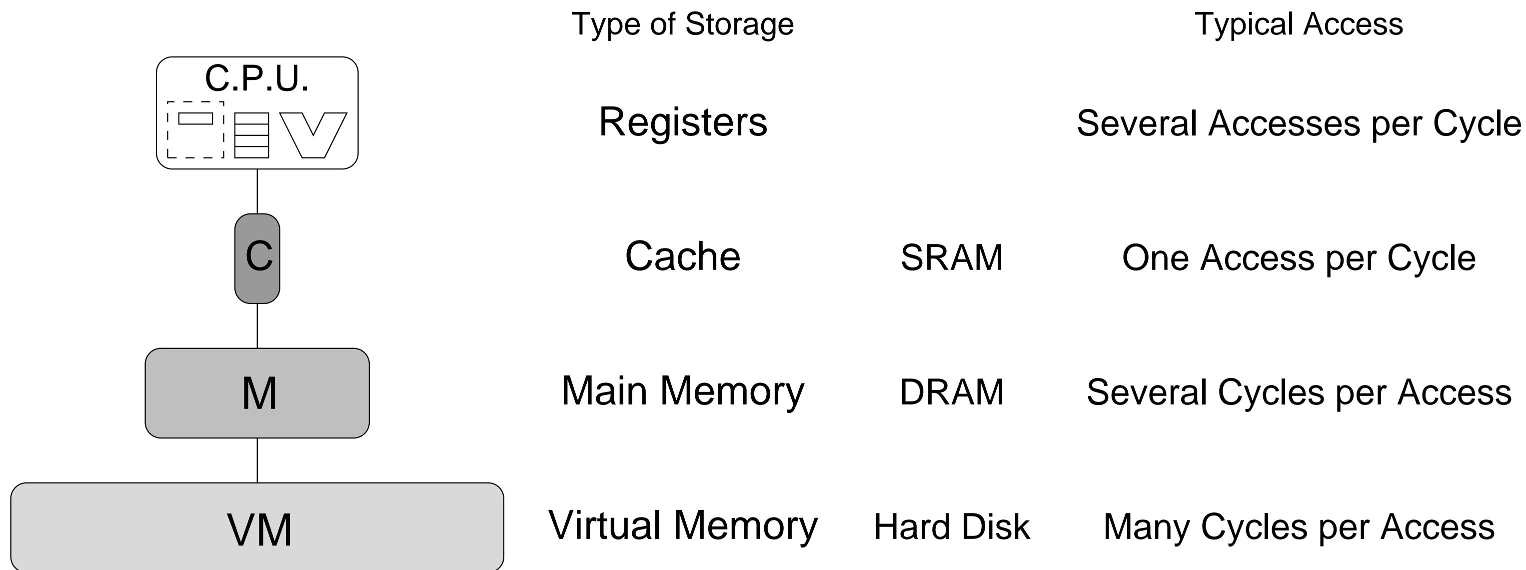
Memory Hierarchy

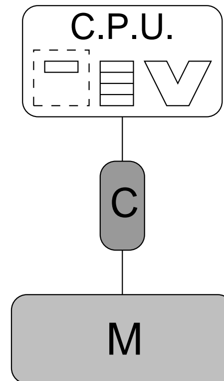
Ideally we would like large amounts of fast memory. This is neither economically viable nor technically possible.



The solution is a hierarchy of memory containing some large components and some fast components.

Our task is to make it appear that we have large amounts of very fast memory.

Cache Memory



Whereas the compiler decides which items are stored in registers for rapid access, the computer must decide which items are stored in the cache.
The compiler will not normally have any knowledge of cache size or organization[1].

Cache memory is based on the *principle of locality*.

[1]This is in contrast to the fast on-chip memory offered by the early Transputers - the compiler knows the size and location of this memory.

Principle of Locality

- *Temporal locality*

  If an item is referenced, it will tend to be referenced again soon.

  – code loops & important data items

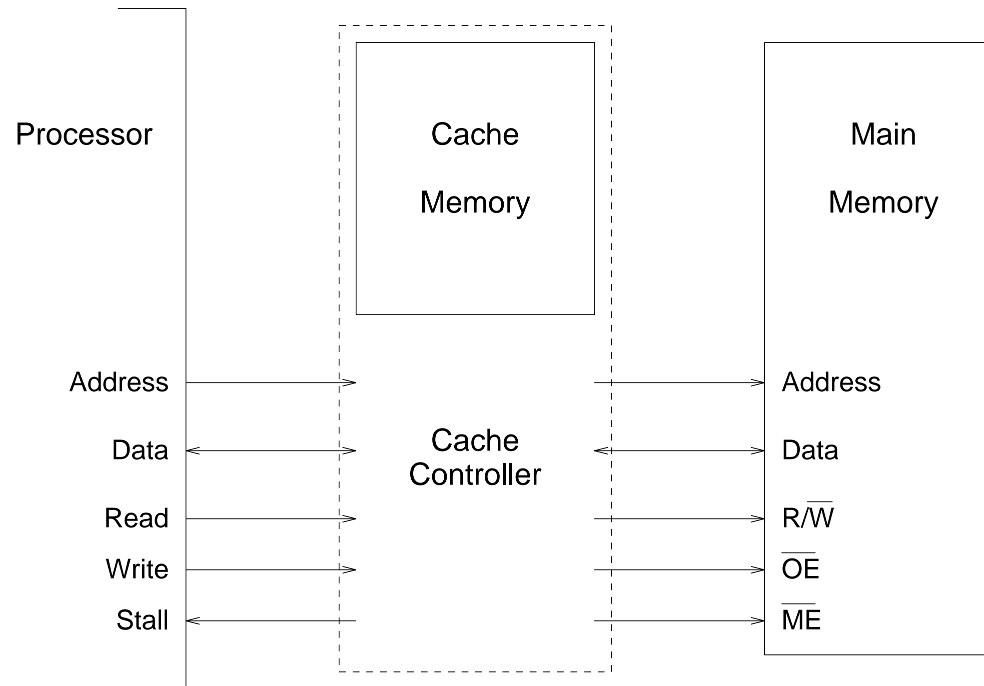
- *Spatial locality*

  If an item is referenced, items whose addresses are close by will tend to be referenced soon.

  – sequential code access & array access

Caching systems exploit this knowledge by storing recently accessed items and their neighbours in fast cache memory.

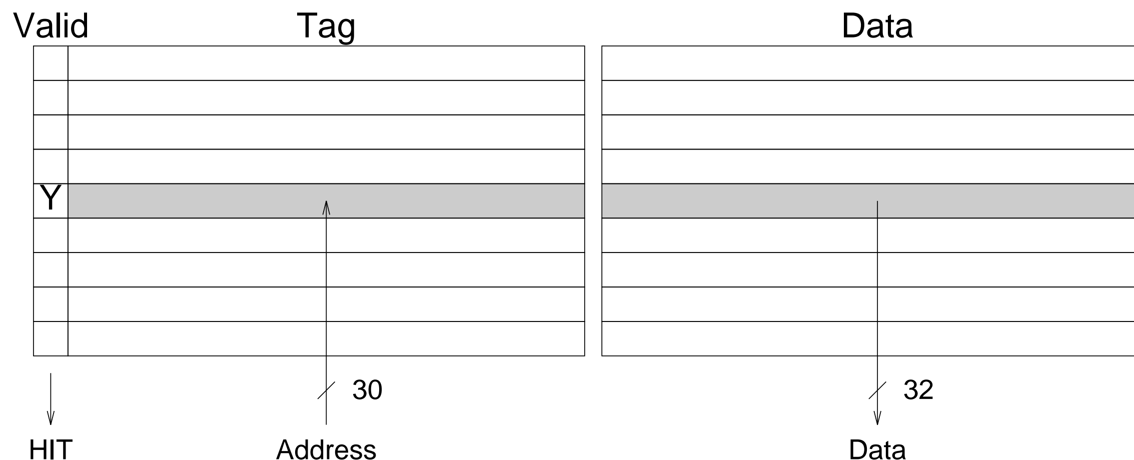
Cache Memory



- The cache controller provides the processor with data from the main memory.
- The cache controller copies accessed items into the cache memory.
- When a copied item is accessed it is retrieved rapidly from the cache memory. Thus the controller makes use of *Temporal locality*.

Ideal Cache

Ideal caches use *content addressable* memories (a.k.a. *associative* memories).



On storage the address and data of an item are stored at any empty location in the memory. The valid flag is set to indicate that the location contains valid data.

On access the requested address is simultaneously compared with each address stored in the *Tag* field of a valid cache location.
If there is a match, the memory flags a *Cache HIT* and returns the appropriate data, otherwise a *Cache MISS* is flagged.

Discard Strategy

Inevitably a finite cache will become full. In order to accept another item the cache must overwrite an old value. The choice of value to overwrite or discard is the discard strategy.

Least Recently Used Discard Strategy

Since we have based the cache on the principle of *temporal locality* it seems reasonable for an ideal cache to discard the item from the cache that has been accessed least recently.

Alternative Cache Types

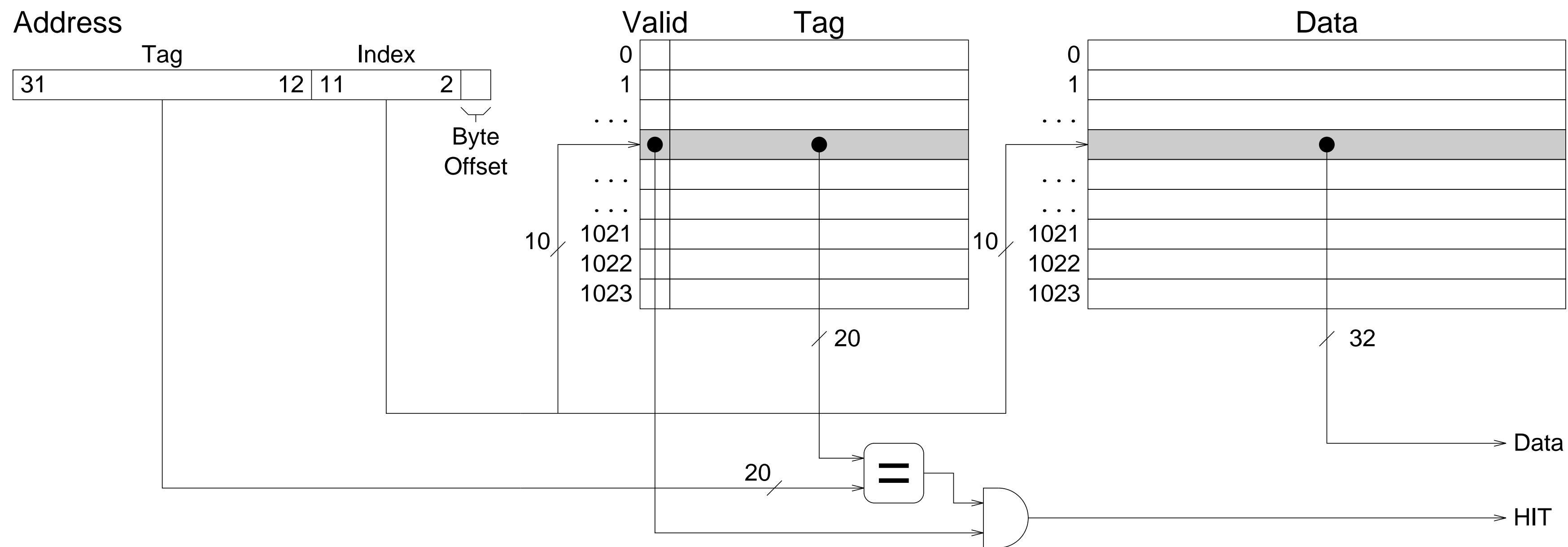
Content addressable memory as used in an ideal cache is not as dense as standard SRAM due the overheads of distributed Tag comparison.

It is possible to achieve much larger cache sizes with caches which can use standard SRAM technology. A larger cache size should increase performance by increasing the *HIT rate*.

Cache types:

- *Fully associative* - ideal cache needs associative memory.
- *Direct mapped* - simple cache using standard SRAM.
- *Set Associative* - compromise cache using standard SRAM.

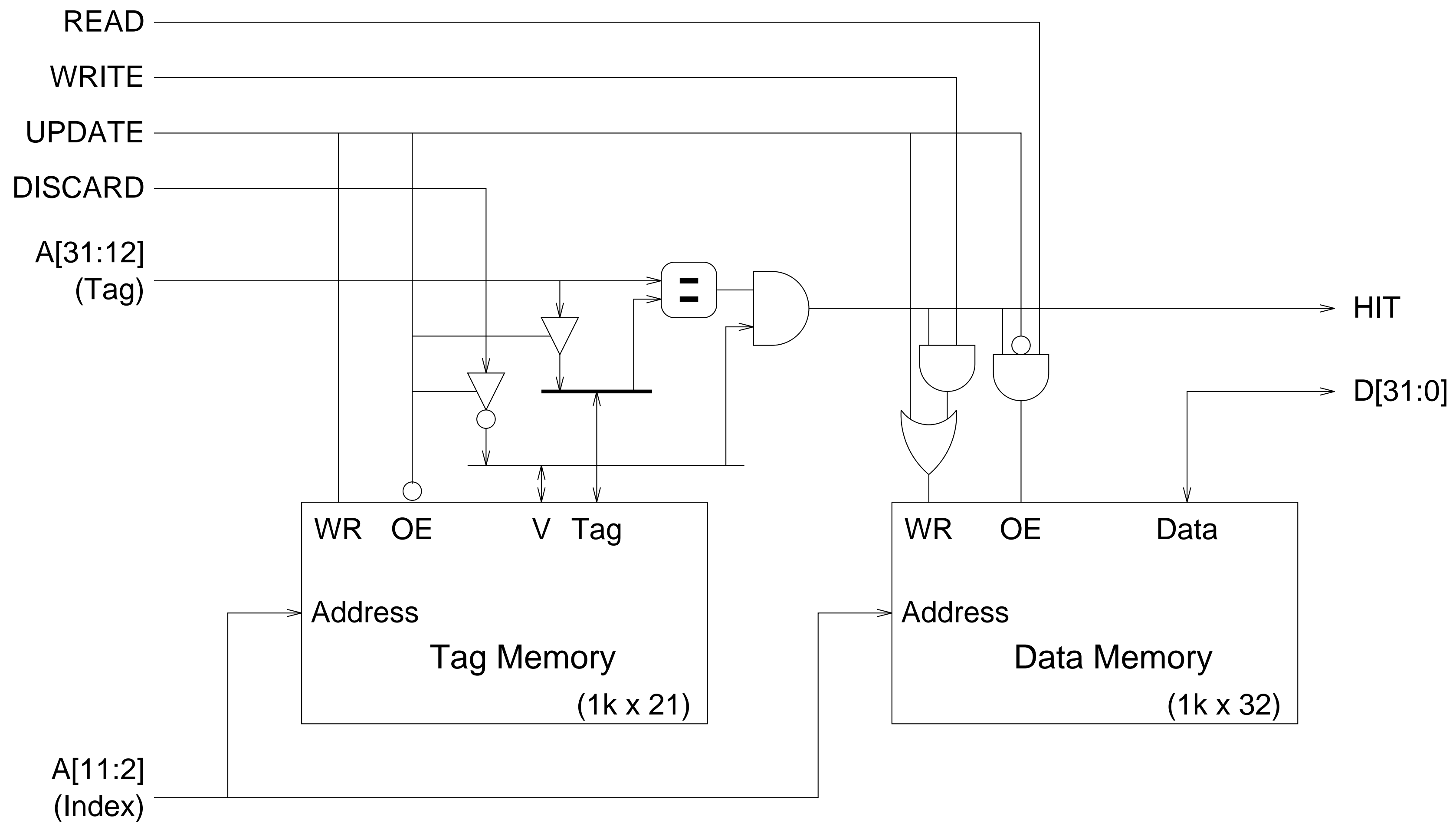
Direct Mapped Cache



- Storage Strategy
  - Cache address (a.k.a. Index) is low order bits of data address
  - The remainder of the data address is stored as the Tag
- Data Recovery
  - Since data can be in only one location data recovery is trivial
  - A simple Tag comparison determines cache HIT/MISS

Direct Mapped Cache

Implementation of direct mapped cache using SRAM components



Strategy

Our implementation:

- Write Strategy
  - *Write Through*

    All write operations update the main memory

    Other caches may use a *Write Back* strategy; data is only written to main memory when it is bumped from the cache.

- Write Miss Strategy
  - *Write Around*

    In the case of a Write Miss we write to the main memory only.

    The alternative of updating the cache for all writes will cause complications for byte and halfword writes.

Write Procedure

- Cache Write (Hit)
  - Write request simultaneous to cache and main memory
  - Hit enables write to cache
  - Address and data are stored at end of first cycle for write to main memory
  - Memory access completes in one cycle – main memory write continues off-line
- Cache Write (Miss)
  - Write request simultaneous to cache and main memory
  - Miss disables write to cache
  - Address and data are stored at end of first cycle for write to main memory
  - Memory access completes in one cycle – main memory write continues off-line

Read Procedure

Read data, copy to cache if not present.

- Cache Read (Hit)
  - Read request simultaneous to cache and main memory
  - Hit enables data from cache and disables data from memory
  - Memory access completes in one cycle
- Cache Read (Miss)
  - Read request simultaneous to cache and main memory
  - Miss disables data from cache and enables data from memory
  - Memory access completes after several cycles
  - In the last cycle an update signal enables a cache *write*

n.b. A read miss, or a write may be additionally delayed where it overlaps with an off-line main memory write.

Cache Performance

$$Average\ memory\ access\ time = Hit\ time + Miss\ rate \times Miss\ penalty$$

- Hit Time

  Time for a cache hit - may be different for read and write.

- Miss Penalty

  Additional time for a cache miss.

- Miss Rate

  In order to analyse miss rate consider the reasons for a miss:

  - Compulsory Miss

    The required block has never been accessed before.

  - Capacity Miss

    The required block was displaced when the cache was full.

  - Collision Miss

    The required block was displaced by another although the cache was not full.

Reducing the Miss Rate

- Capacity Miss
  - Increase cache capacity.
    This simple option will also reduce collision misses.
    The usual limit here is cost although a larger cache may well affect hit time.
- Compulsory Miss
  - Fetch items before they are requested.
    If we can work out which items the CPU will need, we can load them into the cache before a request is made.
    - Pre-fetch Instructions.
      This technique is appropriate for machines with a dedicated memory port for instruction fetch.
    - Pick up an item when it's neighbour is requested.
      This technique is used by caches with increased block size.

Increased Cache Block Size

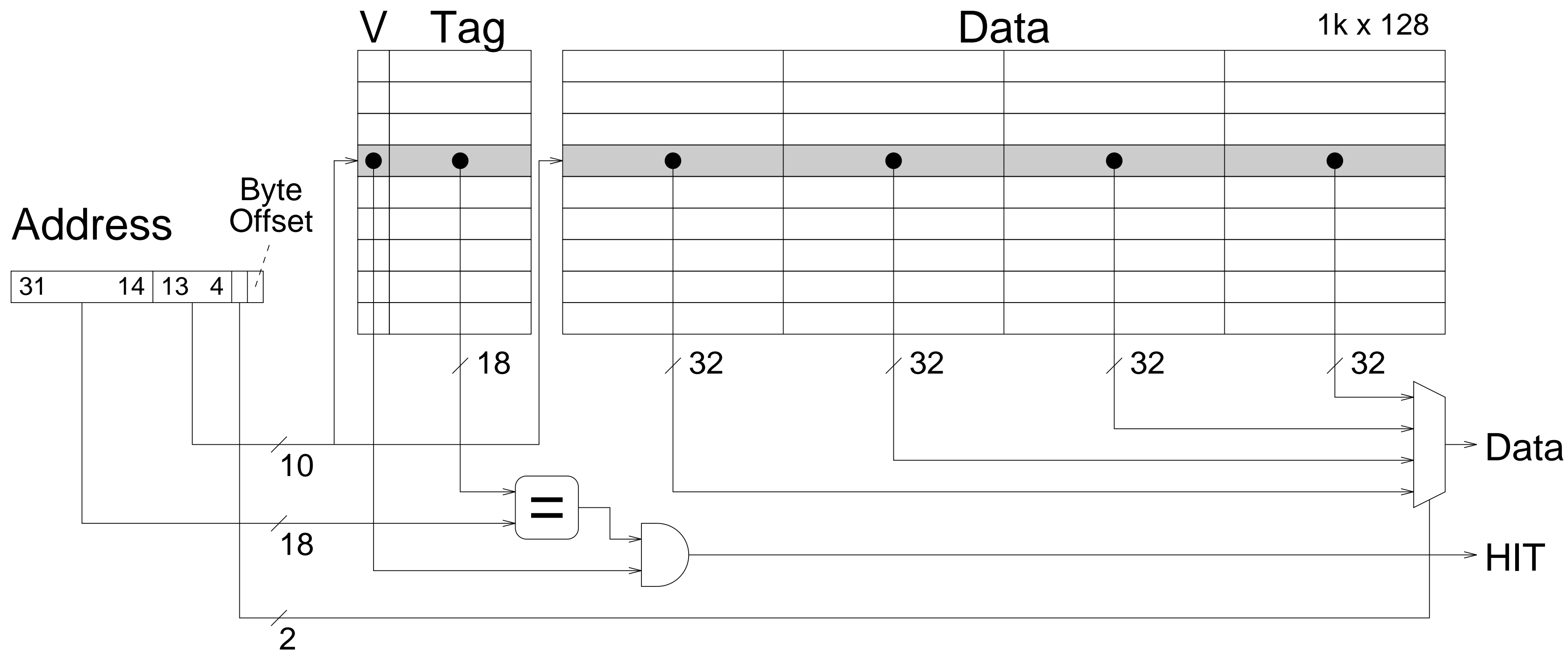
Spatial Locality

In order to take advantage of spatial locality we keep copies of the neighbours of recently used data items.

We increase the block size (a.k.a. line size) of the cache.

The following is a direct mapped cache with a block size of 4 words

Address[3:2] acts as a block offset selecting a word for read/write



Increased Cache Block Size

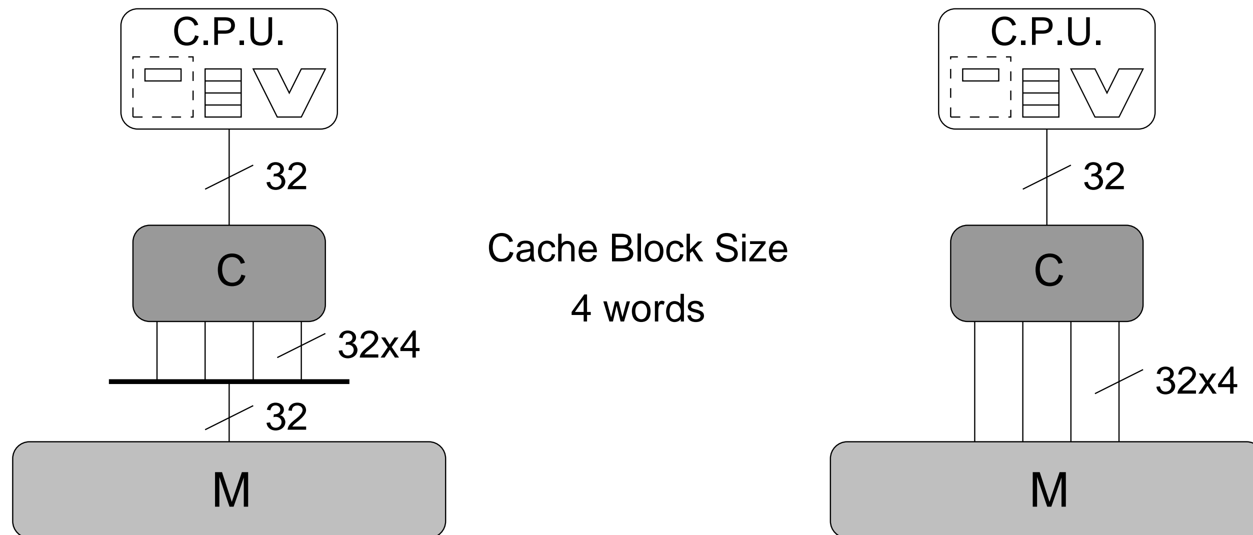
Direct Mapped Cache supporting *Write Through* and *Write Around* strategies:

- Cache Write (Hit)
  - Write Through – single word
- Cache Write (Miss)
  - Write Around – single word
- Cache Read (Hit)
  - Read from cache – single word
- Cache Read (Miss)
  - Read *whole of cache block* from main memory
  - Pass only appropriate word to CPU

Note that a cache with a width of 32 bits appears as a wide cache when the processor makes a byte or halfword access.

Increased Cache Block Size

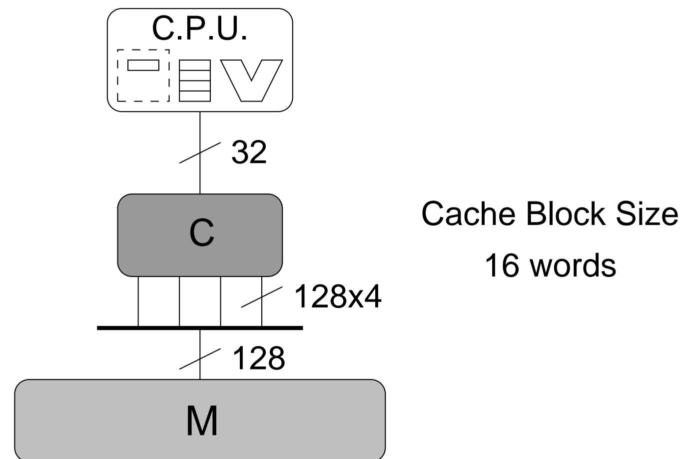
If the path to main memory remains unchanged, the block read miss will take longer to perform.



Alternatively a 128 bit path to memory will increase bandwidth considerably giving much better cache performance.

Increased Cache Block Size

Without matching the block size and the path to memory, there is still a benefit in a reduced miss rate.



The increased miss penalty may be limited by:

- Fast DRAM page mode access.

  The first access to a DRAM row incurs the penalty of row access time.

  Multicolumn accesses to the same row incur this overhead only once.

- Fetch critical word first.

  The critical word is passed to the CPU and the block access completes off-line.

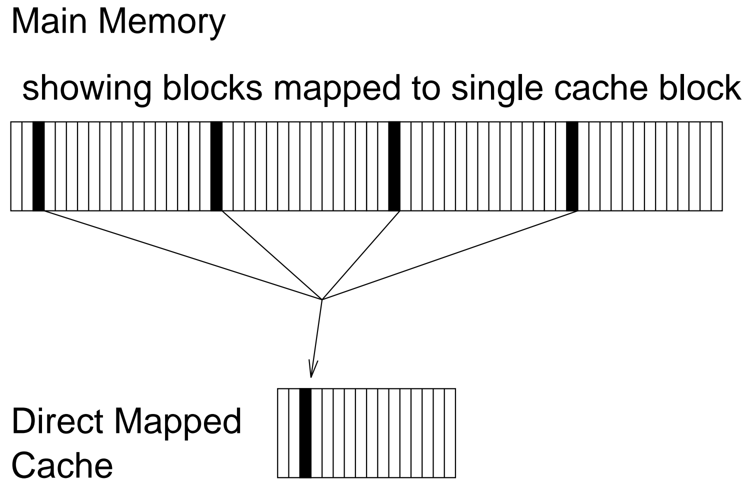
Improving Cache Performance

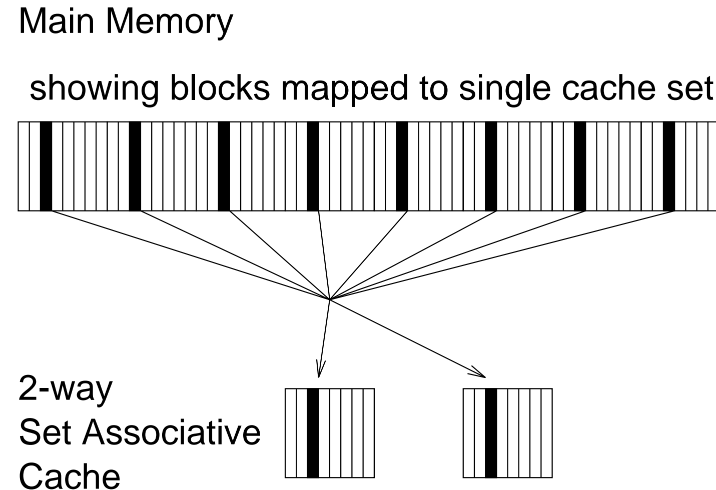
Collision Misses

Given a fixed cache size, the number of collision misses will depend on the number of locations a new block may be placed.

- Fully Associative
  - Block placement – Anywhere
  - No collision misses
- Direct Mapped
  - Block placement – Fixed - many to one mapping
  - Worst situation for collision misses
- Set Associative
  - Block placement – Fixed number of locations for each block
  - Compromise cache with better collision performance than direct mapped, while limiting the task of tag comparison to a small set on each access.

Set Associative Cache





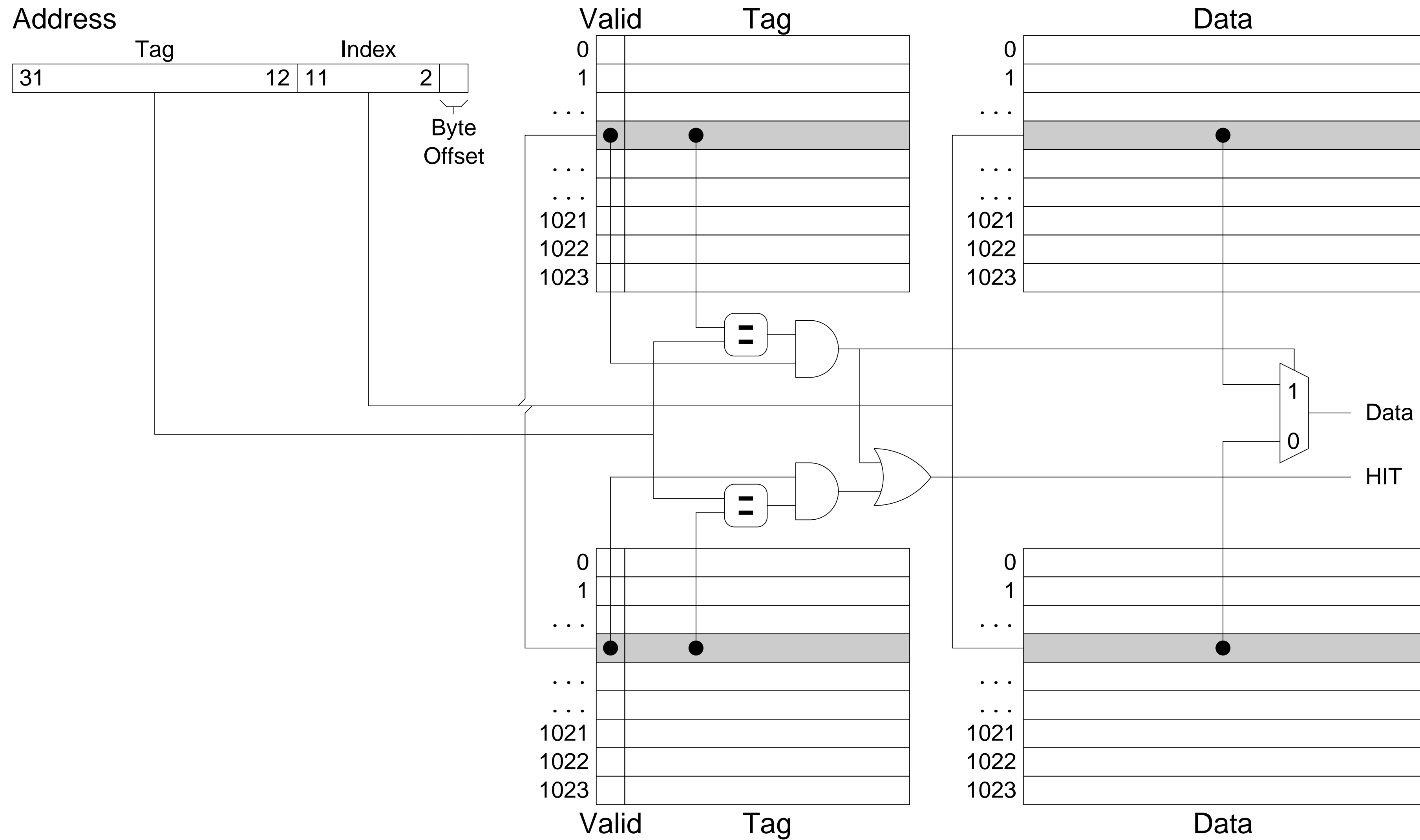
- 2-way Set Associative cache

  A simple two way set associative cache offers two locations for each new block. With only two tag comparisons required per access.

  A collision miss will occur if both locations are full while there is space elsewhere in the cache.

  – The cache may be built from standard SRAM components.

Set Associative Cache



Set Associative Cache

- Cache Write
  - Dual tag comparison selects the appropriate bank for write or indicates a miss.
- Cache Read
  - Dual tag comparison selects the appropriate bank for read or indicates a miss.

    In the case of a miss a decision must be made on where to put the data.

    - - One or more banks indicates no valid data - easy choice.

    - - Otherwise consult discard strategy.

Set Associative Cache

Discard Strategy

- Least Recently Used (LRU)

  We suggested this method for ideal cache. It gives best performance but is rather expensive to implement as the associativity increases.

- Random

  Easy to implement and performs almost as well as LRU as cache size and associativity increase.

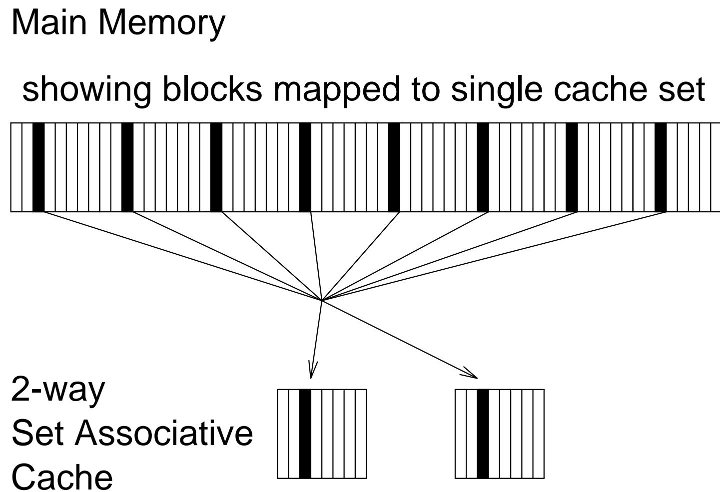
- FIFO

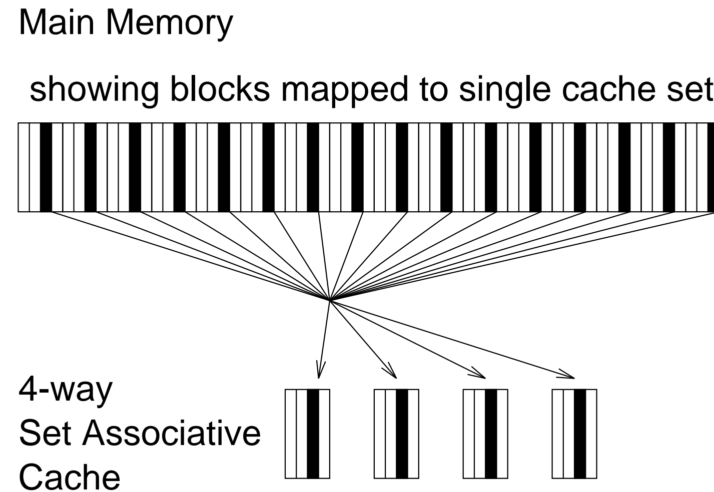
  Although it seems attractive, being easier to implement than LRU, it usually performs less well than random discard.

- None

  Direct mapped caches have no strategy since there is no choice of location for the new block. This is a saving in complexity which should lead to a lower hit time.

Set Associative Cache





- *N*-way Set Associative Cache

  Typically associativity ranges from one (direct mapped) to eight.

  Increased Associativity offers:

  – fewer collision misses

  – greater overhead for tag comparisons, discard strategy etc.

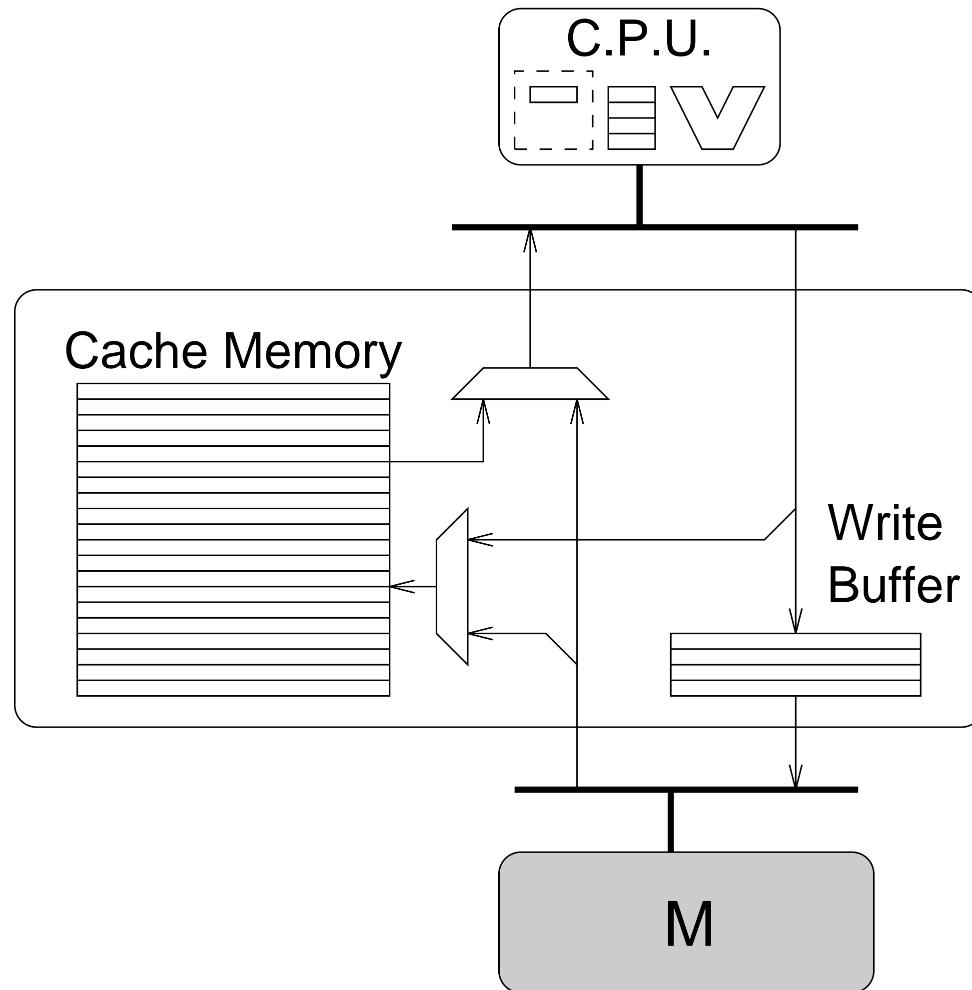
Improving Cache Performance

Reducing the Miss Penalty

In our original analysis only the read miss failed to operate at the speed of the cache.
It was also noted that read miss and write hit/miss might be additionally delayed when they overlap with an off-line write. Wide block reads may also provide such interference.

- Read Miss
  Must increase speed of lower level access.
  - Multi-Level Caches
- Off-line interference
  Must reduce effect of interference
  - Advanced Write Buffer

Write Buffer



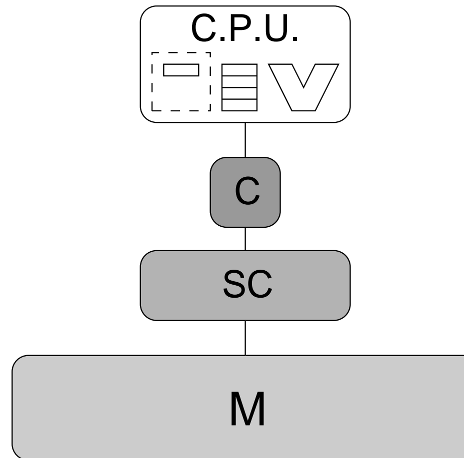
Write Buffer

- Simple Write Buffer
  - single word storage allows write hit/miss to complete in one cycle.

- Advanced features
  - multi-word FIFO buffer
  - merge writes to words within the same block (where main memory is wide)
  - priority of read miss over write
    must check contents of write buffer on a read miss in order to avoid a *read after write* data hazard.

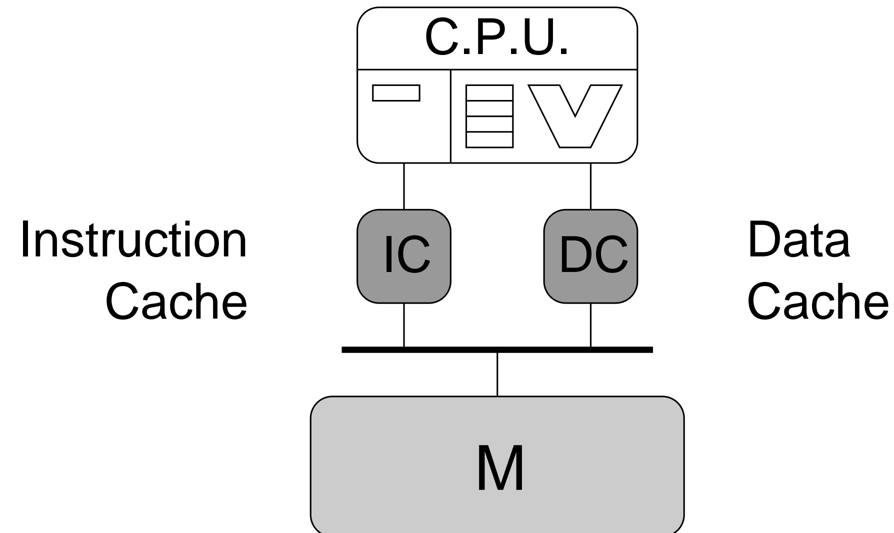
Note a *write back* rather than a *write though* strategy should reduce the number of main memory writes. (Write back is normally used with *fetch on write* write miss strategy which may increase the number of main memory reads.)

Multi-Level Cache



- First Level Cache
  - Simple cache
  - Optimized for hit time
  - Operates at speed of CPU
- Second Level Cache
  - Large cache
  - Optimized for miss rate

Multiple Caches



Pseudo-Harvard Architecture

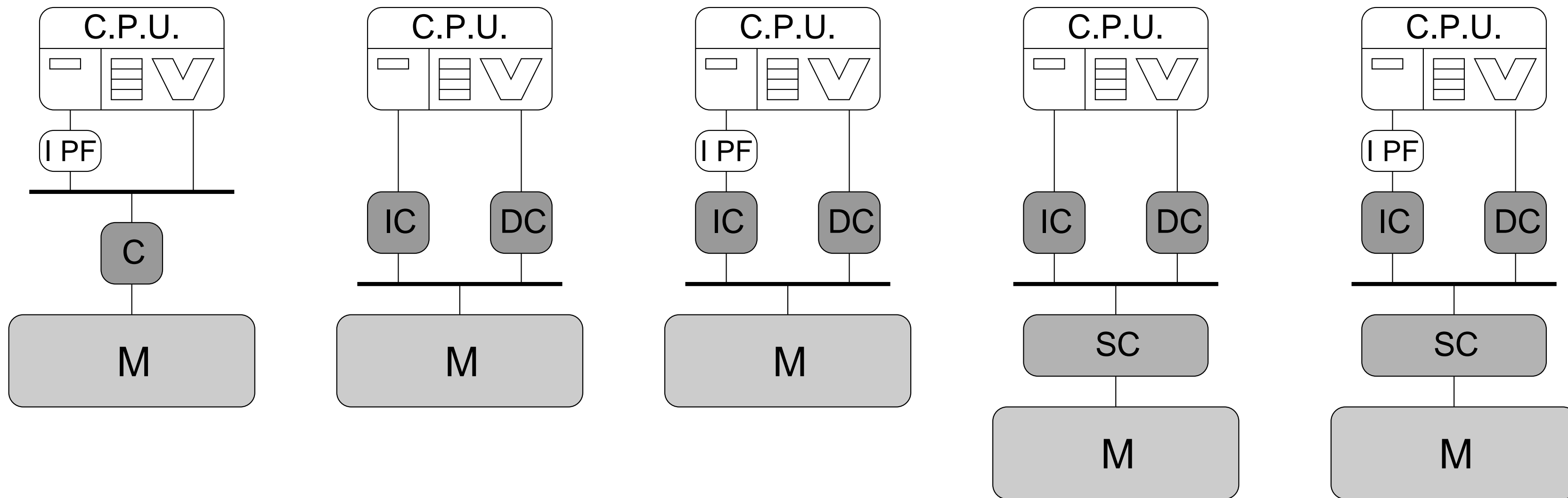
- With multiple caches a machine with an external Princeton architecture may achieve performance approaching that of a Harvard machine.
  - Double CPU to cache bandwidth
  - Separately optimized caches
    Different capacities, block sizes and associativities.
    Instruction caches are read-only with respect to the CPU.

Multiple Caches

- Instruction Pre-Fetch

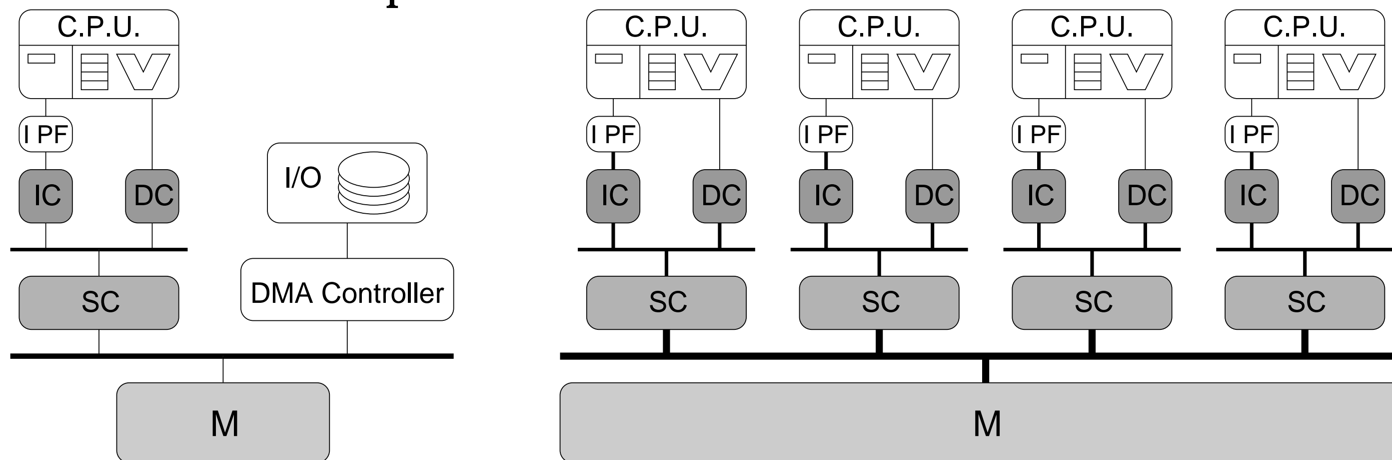
  The pseudo Harvard machine may also make use of an instruction pre-fetch buffer containing one or more blocks of instructions. This buffer reduces the probability of a compulsory miss when the CPU reads a new instruction.

- Cache configurations to support instruction fetch.



Cache Coherence

- Where we have multiple copies of data we must ensure that all copies are the same. This becomes a problem when there are multiple bus masters, e.g. DMA controllers and multiprocessors.

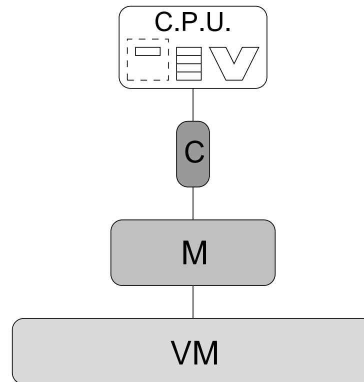


- Bus snooping by cache controllers
  - Discard cache copy?
  - Update cache copy?
- Is it safe to assume that Program cache does not suffer from these problems?

Summary of Cache Features

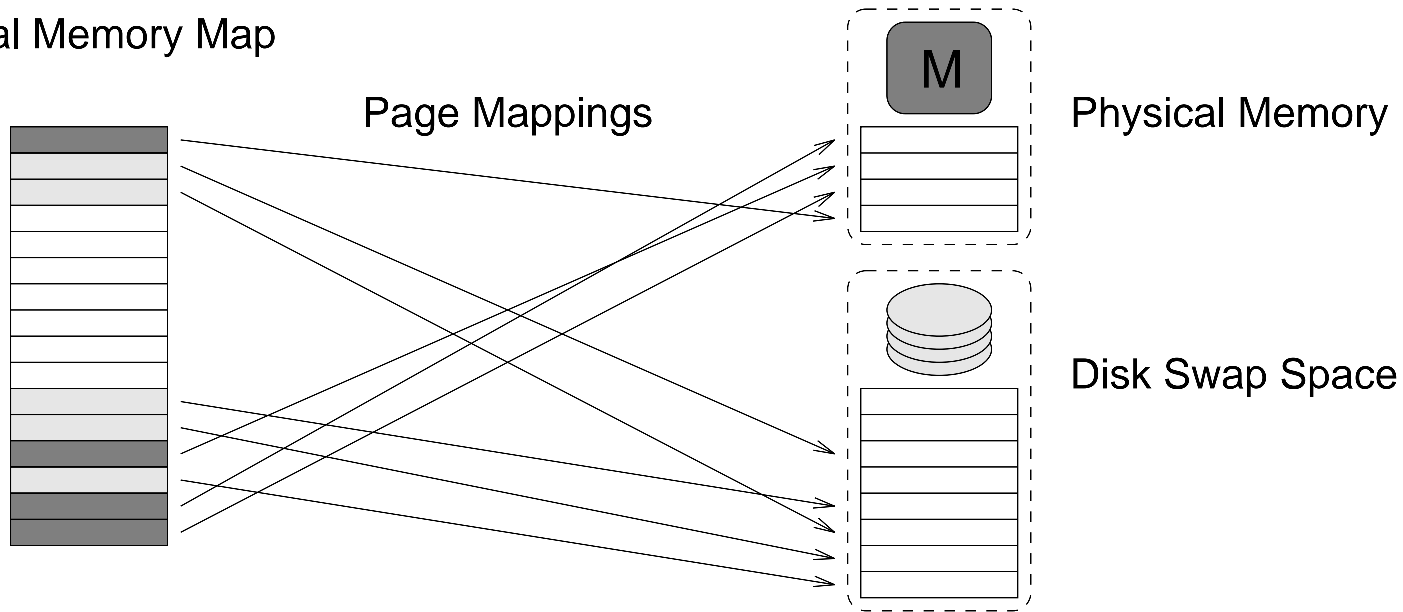
- Capacity
- Block/Line Size
- Storage Strategy
  - Fully Associative – requires associative memory
  - Direct Mapped
  - Set Associative
- Discard Strategy
  - Least Recently Used
  - FIFO
  - Random
  - None – for direct mapped cache
- Write Strategy
  - Write Through
  - Write Around
  - Write Back
  - Fetch on Write

Virtual Memory



- Works like a second level cache
- Large block(page) size
  - miss penalty is horrendous
  - access overhead (seek time) is very long compared to transfer rate.
- Supports *Write Back* and *Fetch on Write*
  - seek time precludes single word *write through* or *write around*
- Supports some form of LRU discard strategy
  - miss penalty justifies this complexity

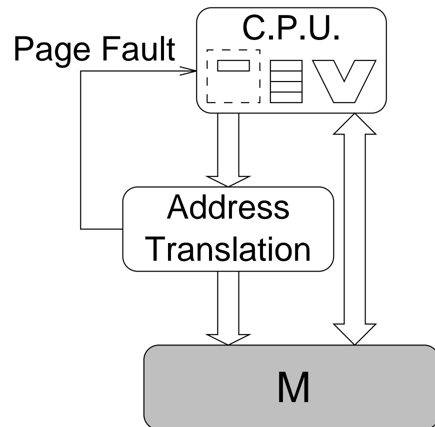
Virtual Memory

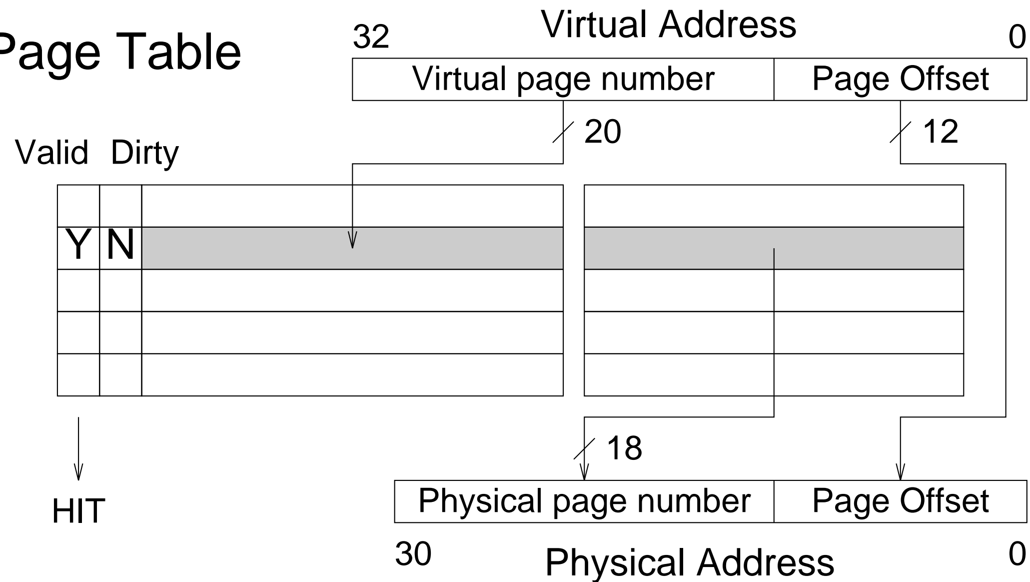


Virtual Memory is broken down into pages:

- Unallocated
- Allocated
  - Mapped to physical memory
  - Swapped out to disk

Virtual Memory





- Page Hit

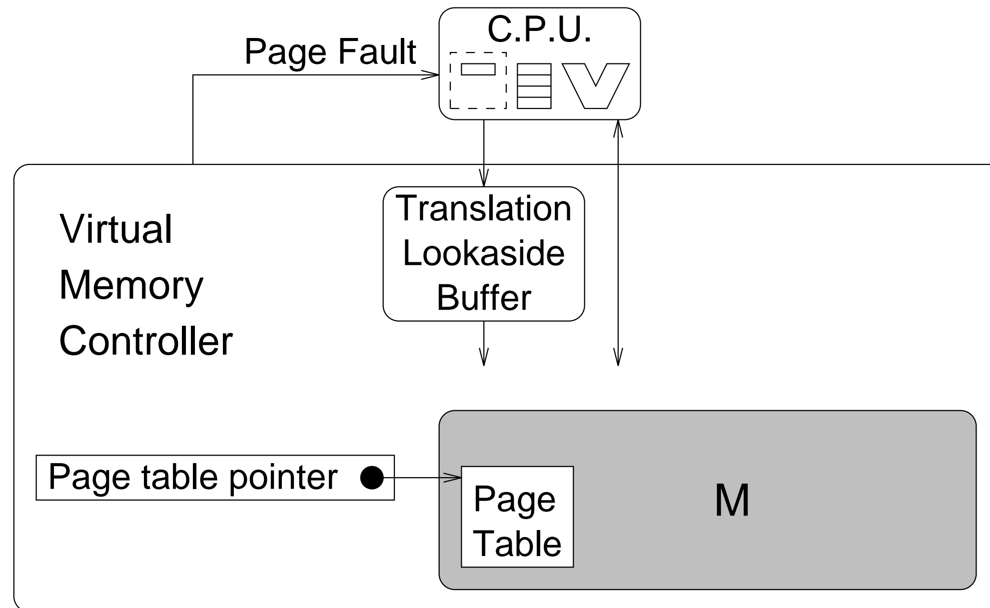
  A fully associative page table provides address translation.

- Page Fault

  An exception is flagged to the processor. Swapping is performed by software[2]. Only *dirty* pages need be copied back to disk when swapped out[3].

[2]compared to the disk seek time the overhead for software swapping is low.
[3]assuming that an old copy is always stored on disk

Virtual Memory



- Due to size restrictions, the page table is usually stored in physical memory.
- A Translation Lookaside Buffer (TLB) acts as a cache for this table.
- A TLB miss triggers the controller to access the full page table.
- A page fault is flagged if the full page table indicates an illegal access[4].

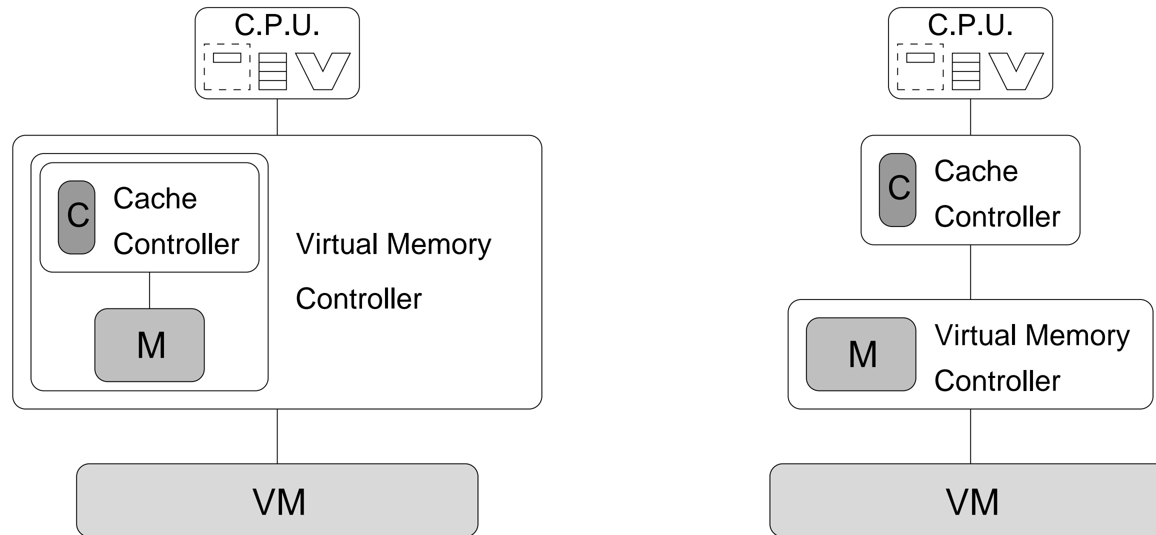
[4] a simpler controller might flag a page fault on TLB miss leaving the CPU to sort out page table access

Memory Management Unit

In fact virtual memory management is usually just one function of the memory management unit. Other functions include:

- DRAM refresh
- Support for multiple processes
  - multiple virtual memory maps
    - - each process exists in its own memory map
    - - each memory map requires a separate page table
    - - the TLB page table cache is flushed when there is a process switch
  - memory protection
    - - page table entries must include read and write permission fields

Virtual Memory



- Cache addresses are physical addresses
  - cache hit time is slowed by page table translation
- Cache addresses are virtual addresses
  - discard cache contents on context switch[5]?

[5] To save cache flush on context switch, the virtual memory address tag must include a field indicating a *context* number. A hit is flagged only if the virtual address and the current context match the values stored in the tag. The same system can be used to avoid TLB flush.