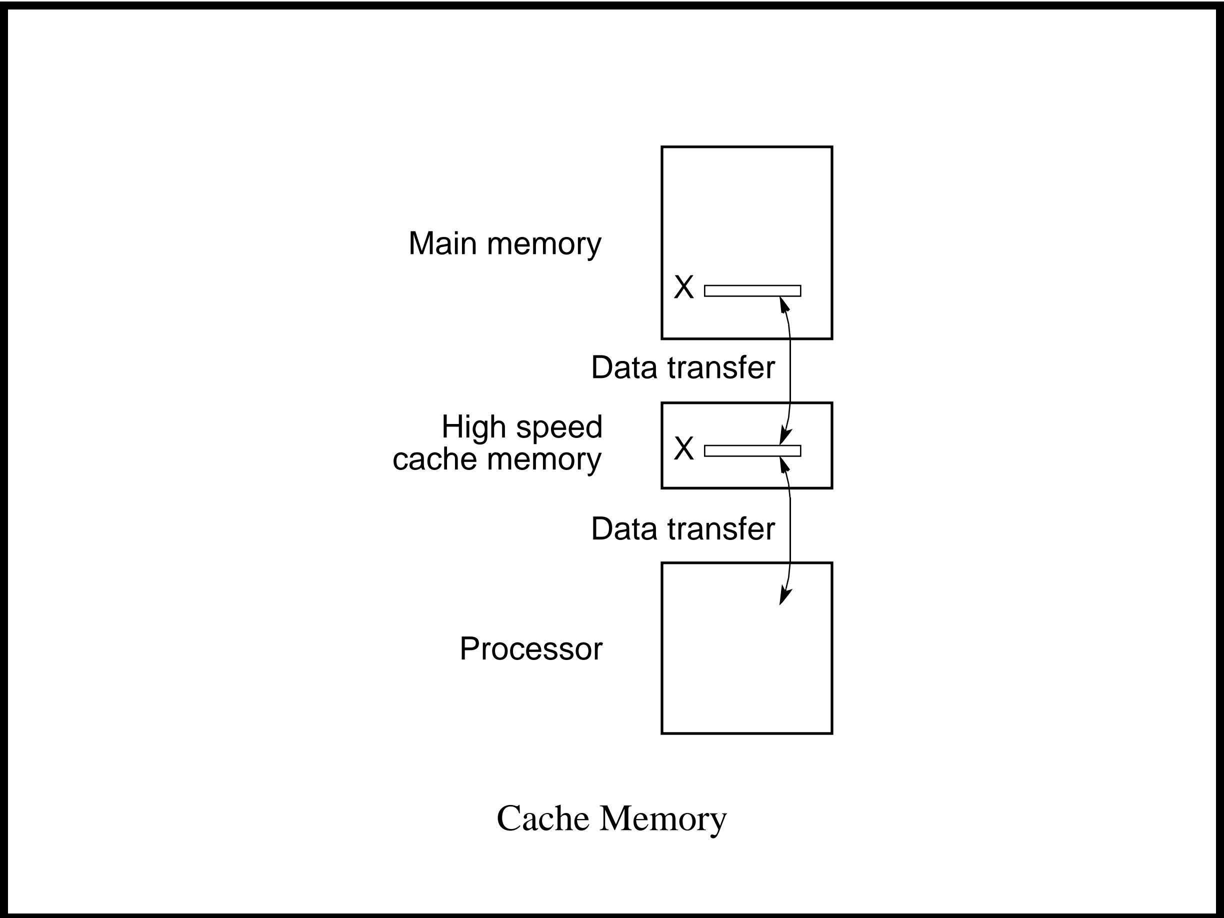
Memory Design

Cache Memory

Processor operates much faster than the main memory can.

To ameliorate the sitution, a high speed memory called a cache memory placed between the processor and main memory.

Barry Wilkinson 2000. All rights reserved.



Cache Memory

Memory access time – the time between the submission of a memory request and the completion of transfer of information into or from the addressed location.

Normally, access time for read and write operations the same, and we will assume this.

Memory cycle time – the minimum time that must elapse between two successive operations to access locations in the memory (read or write).

If programs were executed purely in sequence, from one memory address onwards, and the same instructions were never re-executed, caches would cause an additional overhead as information would first have to be transferred from the main memory to the cache and then to the processor and vice versa. The access time would then be:

$$t_a = t_m + t_c$$

where t_c = cache access time and t_m = main memory access time.

Fortunately though code is generally executed sequentially, virtually all programs repeat sections of code and repeatedly access the same or nearby data. This characteristic is embodied in the *Principle of Locality*.

Principle of Locality

Found empirically to be obeyed by most programs. Applies to both instruction references and data references, though more likely in instruction references.Two aspects:

1. *Temporal locality* (locality in time) – individual locations, once referenced, are likely to be referenced again in the near future.
2. *Spatial locality* (locality in space) – references, including the next location, are likely to be near the last reference. (References to the next location are sometimes known as *sequential locality*.)

Barry Wilkinson 2000. All rights reserved.

Temporal locality is found in instruction loops, data stacks and variable accesses

Spatial locality describes the characteristic that programs access a number of distinct regions. Sequential locality describes sequential locations being referenced and is a main attribute of program construction. It can also be seen in data accesses, as data items are often stored in sequential locations.

Temporal locality is essential for an effective cache.

Spatial locality helpful when we design a cache as we shall see, but it is not essential.

Taking advantage of temporal locality

Suppose a memory reference is repeated *n* times in all during a program loop and, after the first reference, the location is always found in the cache, then the average access time would be:

$$\text{Average access time} = \frac{(nt_c + t_m)}{n} = t_c + \frac{t_m}{n}$$

where n = number of references.

Barry Wilkinson 2000. All rights reserved.

Example

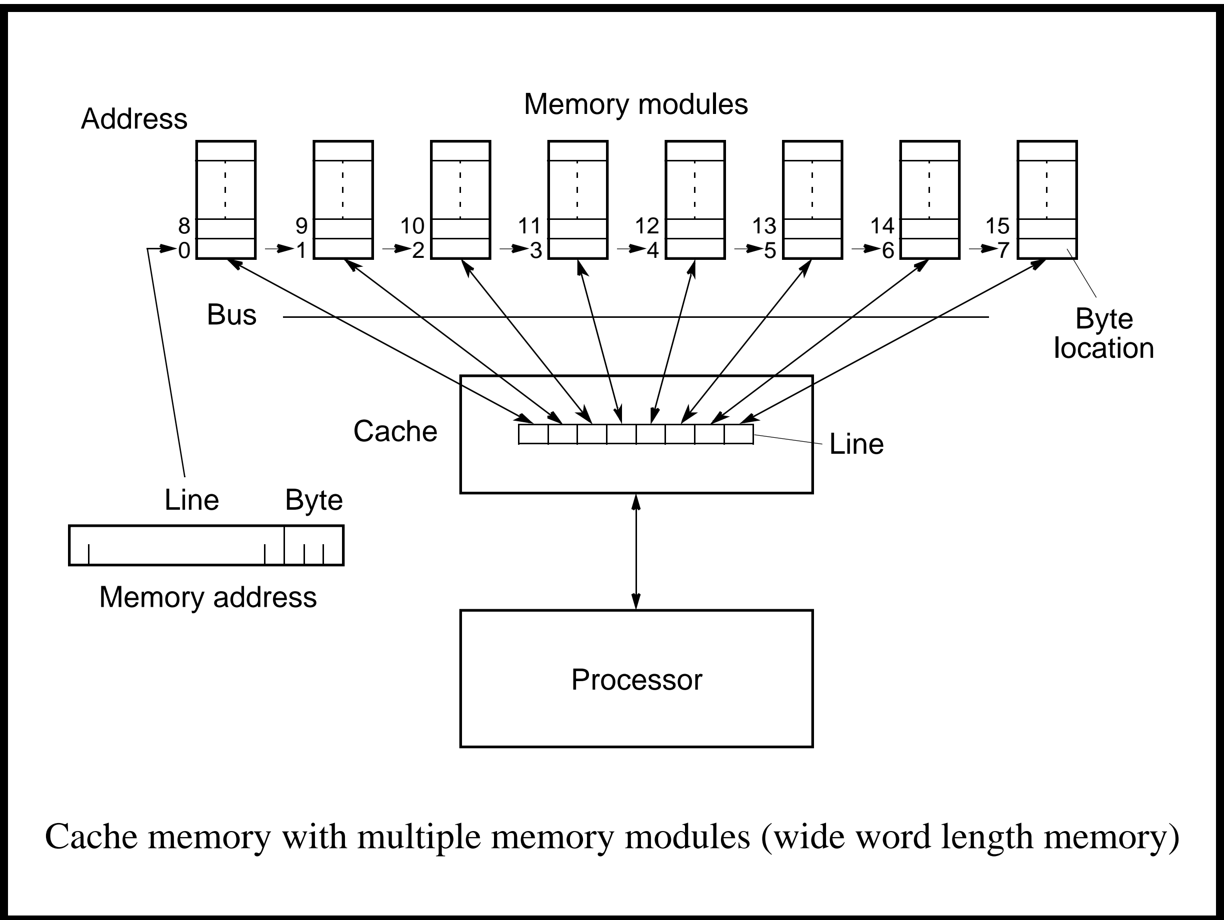
If $t_c = 5$ ns, $t_m = 50$ ns and $n = 10$, the average access time would be 10 ns, as opposed to 50 ns without the cache.

Taking advantage of spatial locality

To take advantage of spatial locality, we will transfer not just one byte or word from the main memory to the cache (and vice versa) but a series of sequential locations called a *line* or a *block*. (Both terms are used in the literature – we shall use the term line.)

For best performance, the line should be transferred simultaneously across a wide data bus to the cache, with one byte or word being transferred from each memory module. This also enables the access time of the main memory to be matched to the cache.

Barry Wilkinson 2000. All rights reserved.



Cache memory with multiple memory modules (wide word length memory)

Number of memory modules, m say, is chosen to produce a suitable match in the speed of operation of the main and cache memories.

For a perfect match, m would be chosen such that $mt_c = t_m$.

The cache words could subsequently be accessed by the processor in sequential order from the cache in another mt_c seconds.

Barry Wilkinson 2000. All rights reserved.

The average access time of these words when first referenced would be

$$2mt_c/m = 2t_c.$$

Should the words be referenced n times in all, the average access time would be:

$$\text{Average access time} = \frac{2t_c + (n-1)t_c}{n} = \frac{(n+1)t_c}{n}$$

Example

If a cache has an access time of 5 ns and the main memory has an access time of 50 ns, eight main memory modules would allow eight words to be transferred to the cache in 200 ns, or an average of 200/8 ns per word. With ten references in all, we have:

$$\text{Average access time} = \frac{(25 + 9 \times 5)}{10} = 7 \text{ ns}$$

The average access time is approximately t_c for large n..

Barry Wilkinson 2000. All rights reserved.

Hit Ratio

– the probability that the required word is already in the cache. A *hit* occurs when a location in the cache is found immediately, otherwise a *miss* occurs and a reference to the main memory is necessary.

The cache *hit ratio*, h, (or *hit rate*) is defined as:

$$h = \frac{\text{Number of times required word found in cache}}{\text{Total number of references}}$$

The *miss ratio* (or *miss rate*) is given by 1 – h.

Average access time

Average access time, t_a, given by:

$$t_a = t_c + (1 - h)t_m$$

assuming again that the first access must be to the cache before an access is made to the main memory. Only read requests are consider so far.

Example

If hit ratio is 0.85 (a typical value), main memory access time is 50 ns and cache access time is 5 ns, then average access time is $5+0.15\times50 = 12.5$ ns.

THROUGHOUT t_c is the time to access the cache, get (or write) the data if a hit or recognize a miss. In practice, these times could be different.

Barry Wilkinson 2000. All rights reserved.

Alternative equation

$$t_a = t_{hit} + (1 - h)t_{miss_pen}$$

where t_{hit} = the time to access the data should be it in the cache (the hit time) and t_{miss_pen} is the extra time require if the data is not in the cache (the *miss penalty*), i.e.

Averge memory access time = hit time + miss rate × miss penalty

This form is used by Hennessy and Patterson (1996).

Our equations can be put in this form
by simply substitution t_{hit} for t_c and $t_{miss\text{-}pen}$ for t_m

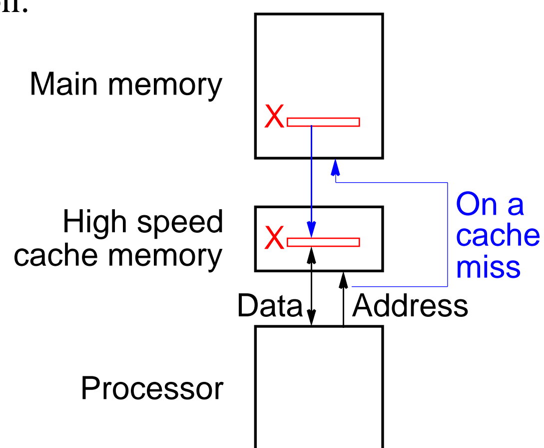
In a practical system, each access time will be given as an integer number of machine cycles This can be applied to all equations. For example, in the previous equation, typically the hit time will be 1–2 cycles. The cache miss penalty is often in the order of 5–20 cycles.

Cache Memory Organizations

Need a way to select the location within the cache. The memory address of its location in main memory is used.

Three ways of selecting cache location:

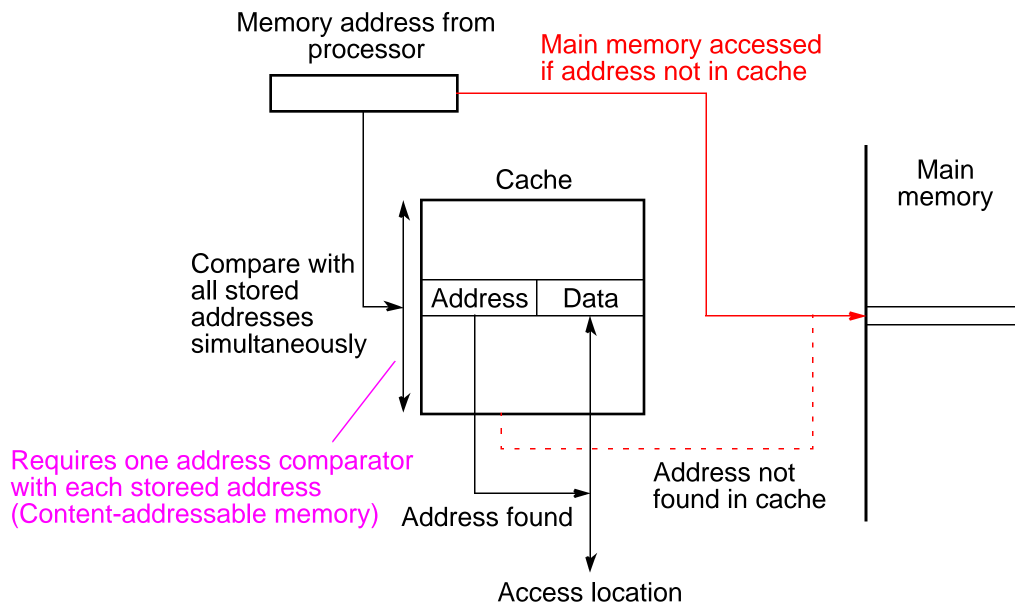
1. Fully associative
2. Direct mapped
3. Set associative



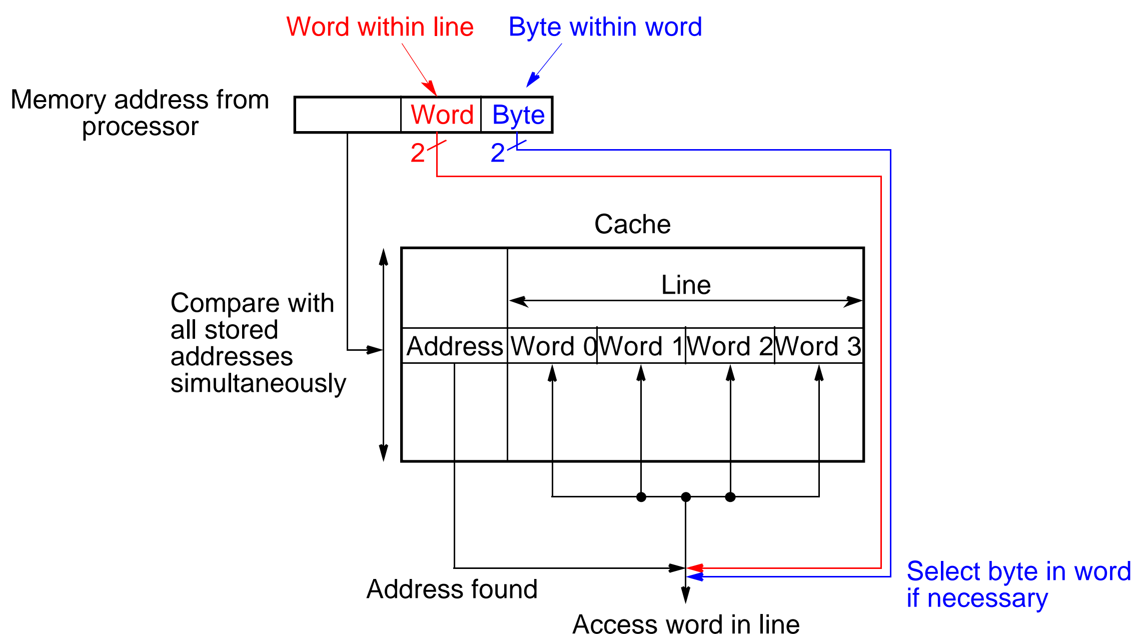
Barry Wilkinson 2000. All rights reserved.

Fully Associative Mapping

Both memory address and data stored together in the cache. Incoming memory address is simultaneously compared with all stored addresses using the internal logic of the cache memory.



Caches organize their stored information into groups of consecutive bytes called lines (or blocks). Each line could be say 16 bytes. With 32-bit processors, we might need to access a word consisting of 4 bytes:



Barry Wilkinson 2000. All rights reserved.

Fully associative cache needs an algorithm to select where to store information in cache, generally over some existing line (which would have to be copied back to the main memory if altered).

Must be implemented in hardware. (No software)

The replacement algorithm should choose a line which is not likely to be needed again in the near future, from all the lines that could be selected.

Common Algorithms

1. Random selection
2. The least recently used algorithm (or an approximation to it).

Barry Wilkinson 2000. All rights reserved.

Least Recently Used (LRU) Algorithm

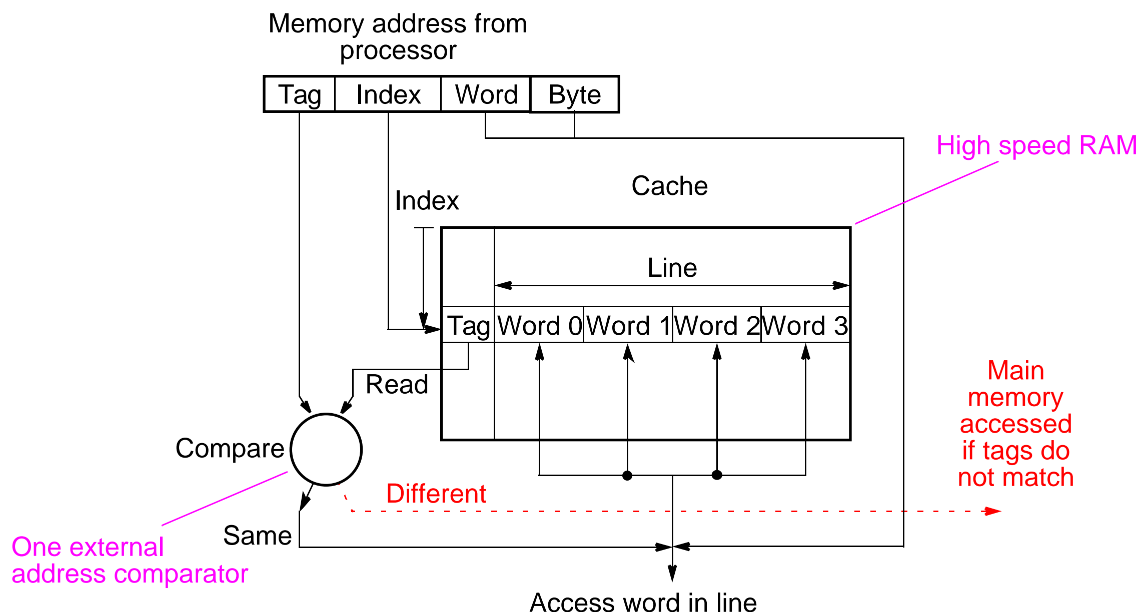
The line which has not been referenced for the longest time is removed from the cache.

The word “recently” comes about because the line is not the least used, as this is likely to be back in memory. It is the least used of those lines in the cache, and all of these are likely to have been recently used otherwise they would not be in the cache.

Can only be implemented in hardware fully when the number of lines that need to be considered is small (see later).

Direct Mapping

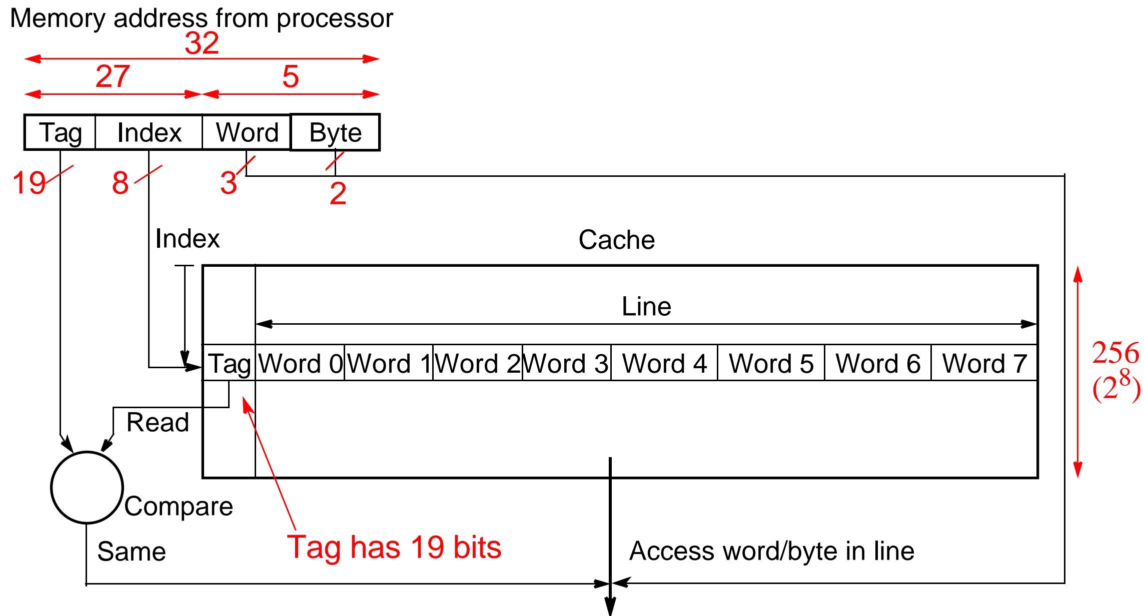
Data held in cache at an address given by lower significant bits of main memory address. Line selected from lower significant bits of memory address. Remaining higher significant bits of address stored in cache:



Barry Wilkinson 2000. All rights reserved.

Sample Direct-Mapped Cache Design

8192-byte direct mapped cache with 32-byte line organized as eight 4-byte words. 32-bit memory address.



Advantages of Direct Mapped Caches

1. No replacement algorithm necessary.
2. Simple hardware and low cost.
3. High speed of operation.

Major Disadvantage

Performance drops significantly if accesses are made to different locations with the same index.

(As the size of cache increases, the difference in the hit ratios of the direct and associative caches reduces and becomes insignificant.)

Barry Wilkinson 2000. All rights reserved.

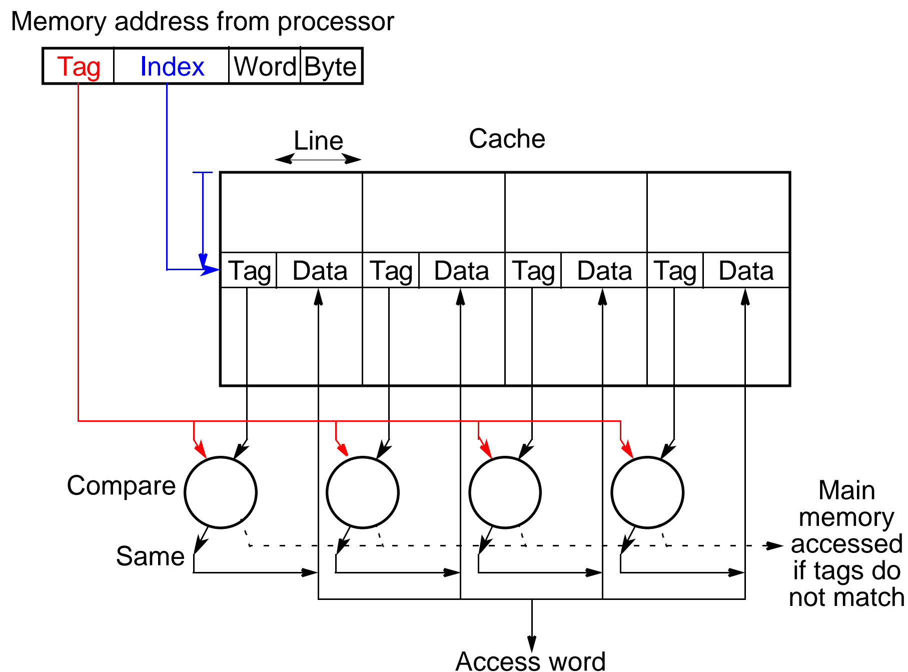
Set-Associative Mapping

Allows a limited number of lines, with the same index and different tags, in the cache. A compromise between a fully associative cache and a direct mapped cache.

Cache divided into "sets" of lines. A four-way set associative cache would have four lines in each set.

The number of lines in a set is known as the *associativity* or set size. Each line in each set has a stored tag which, together with the index (set number), completes the identification of the line.

4-way Set-Associative Cache



First, index of address from processor used to access set. Then, all tags of selected set compared with incoming tag. If match found, corresponding location accessed, otherwise access main memory.

Barry Wilkinson 2000. All rights reserved.

Set-Associative Cache Replacement Algorithm

Need only consider the lines in one set, as the choice of set is predetermined by the index in the address.

Typically, the set size is 2, 4, 8, or 16. A set size of one line reduces the organization to that of direct mapping and an organization with one set becomes fully associative mapping.

Set-associative cache has been popular for internal caches of microprocessors. Examples: Motorola MC68040 (4-way set associative), Intel 486 (4-way set associative), Intel Pentium (two-way set associative).

Valid Bits

In all caches, at least one a valid bit is provided in the cache with each line. Valid bit set to a 1 when the contents of the line is valid. Checked before accessing line. Needed to handle the start-up situation when a cache will hold random patterns of bits.

Barry Wilkinson 2000. All rights reserved.

Valid Bits continued

In some systems, size of the line is larger than data path between the cache and the memory so that only part of a line that can be transferred between the main memory in one transfer. Multiple transfers will be necessary to fill the line from the main memory and to transfer altered lines back to the main memory.

It is possible for the line not to hold all the words associated with that line during the period that words are being transferred into the cache one after the other; some words might be from a previous line. Then a valid bit is needed which each part of the line.

Fetch policy

Three strategies for fetching bytes or lines from the main memory to the cache:

1. Demand fetch.
2. Prefetch.
3. Selective fetch.

Barry Wilkinson 2000. All rights reserved.

Demand fetch - fetching a line when it is needed on a miss.

Prefetch - fetching lines before they are requested.

Simple prefetch strategy - prefetch (i + 1)th line when ith line is initially referenced (assuming that the (i + 1)th line is not already in the cache) on the expectation that it is likely to be needed if the ith line is needed.

Selective fetch - policy of not always fetching lines, dependent upon some defined criterion. Then, the main memory rather than the cache to hold the information. Individual locations could be tagged as non-cacheable.
May be advantage to *lock* certain lines so that these are not be replaced. Hardware could be provided within the cache to implement such locking.

Write Operations

As reading a word in the cache does not affect it, no discrepancy between the cache word and copy held in main memory.

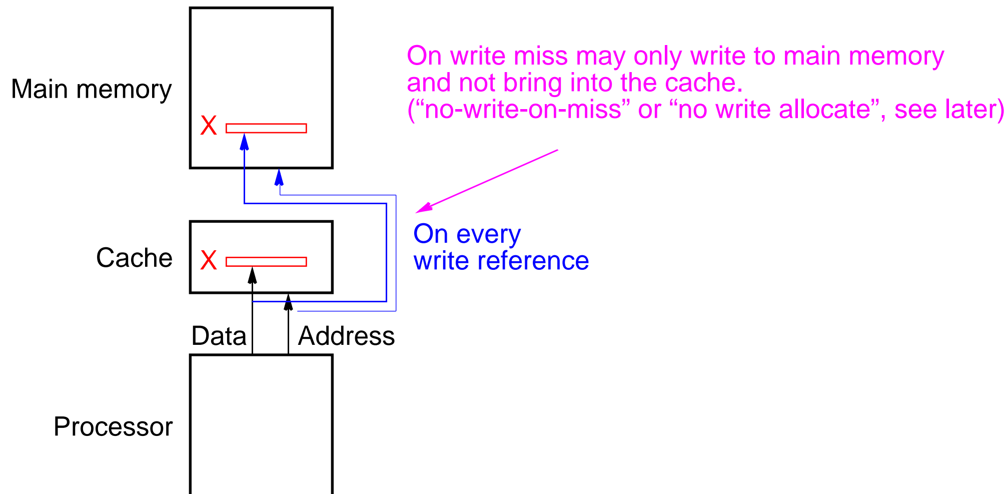
Writing can occur to cache words and then copy held in main memory different.

Two principal alternative mechanisms to update the main memory:

- Write through
- Write back

Write-Through

In the *write-through* mechanism, every write operation to the cache is repeated to the main memory, normally at the same time.



Write-through equations

With transfers from main memory to the cache on all misses (read and write):

$$t_a = t_c + (1 - h)t_m + w(t_m - t_c)$$

where t_m = time to transfer line to cache, assuming the whole line must be transferred (and it can be done in one transaction) and w = fraction of write references.

$(t_m - t_c)$ is additional time to write word to main memory whether hit or miss, given that both cache and main memory write operations occur simultaneously but main memory write must complete before subsequent cache operation can proceed.

Barry Wilkinson 2000. All rights reserved.

Example

Suppose $t_c = 25$ ns, $t_m = 200$ ns, h = 99 per cent, $w = 20$ per cent, and the memory data path fully matches the cache line size. The average access time would be 62 ns.

Multiple transfers to transfer line

If line is longer than the external data path, separate data transfers are needed for each word of a line. Then

$$t_a = t_c + (1 - h)t_b + w(t_m - t_c)$$

where $t_b = bt_m$ and there are b transfers to transfer the complete line.

This modification applies to all the equations.

Barry Wilkinson 2000. All rights reserved.

Fetch-on-write (miss)

Describes a policy of bringing a word/line from the main memory into the cache for a write operation.

No-Fetch-on-write (miss)

Describes a policy of not bringing a word/line from the main memory into the cache for a write operation.

The term *allocate on write* is sometimes used for fetch on write because a line is allocated for an incoming line on cache miss. *Non-allocate on write* corresponds to no fetch on write.

No-Fetch-on-write equation for write-through cache

The average access time with a no fetch on write policy is given by:

$$t_a = t_c + (1 - w)(1 - h)t_m + w(t_m - t_c)$$

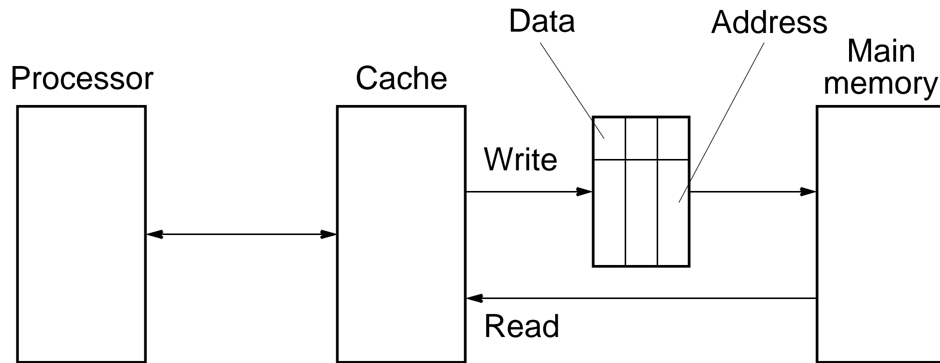
The hit ratio will generally be slightly lower than for the fetch on write policy because altered lines will not be brought into the cache and might be required during some read operations, depending upon the program.

No fetch on write often practiced with a write-through policy. Why?

Barry Wilkinson 2000. All rights reserved.

Cache with write buffer

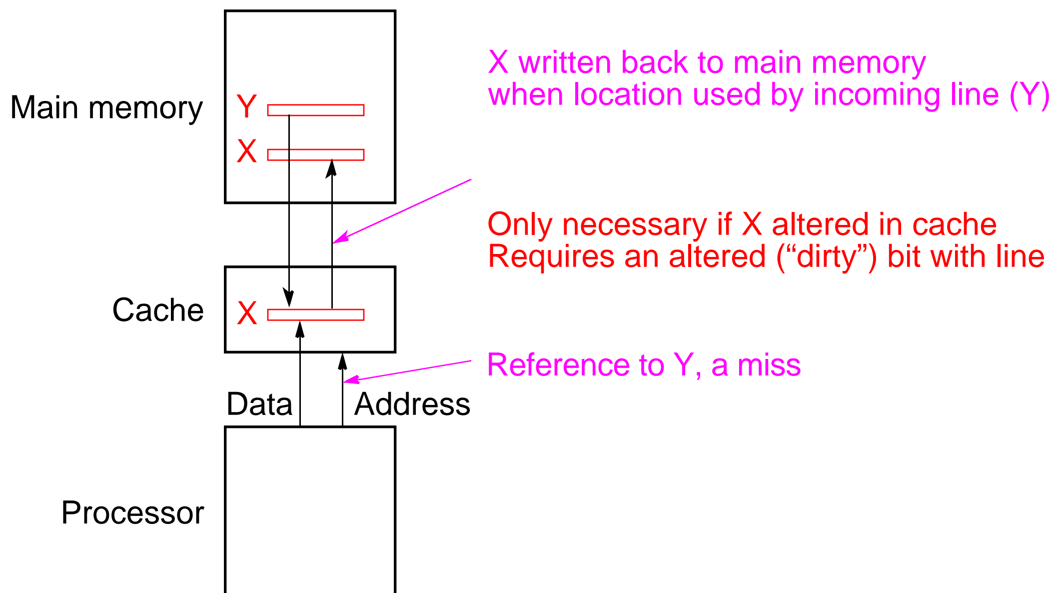
The write-through scheme can be enhanced by incorporating buffers:



Allows the cache to be accessed while multiple previous memory write operations proceed. “Non-blocking” cache.

Write-Back (or copy back)

Write operation to main memory only done at line replacement time. At this time, line displaced by incoming line written back to main memory.



Barry Wilkinson 2000. All rights reserved.

Write-Back Equations

"Simple" write-back:

$$t_a = t_c + (1 - h)t_m + (1 - h)t_m = t_c + 2(1 - h)t_m$$

One $(1 - h)t_m$ for writing a line back to memory, other $(1 - h)t_m$ for fetching a line from memory.

Write-back normally handles write misses as fetch on write. Why?

Write-Back with write back of modified lines

Write-back mechanism usually only writes back lines that have been altered. The average access time now becomes:

$$t_a = t_c + (1 - h)t_m + w_b(1 - h)t_m = t_c + (1 - h)(1 + w_b)t_m$$

where w_b is the probability that a line has been altered (fraction of lines altered).

The probability that a line has been altered could be as high as the probability of write references, w, but is likely to be much less, as more than one write reference to the same line is likely and some references to the same byte/word within the line likely.

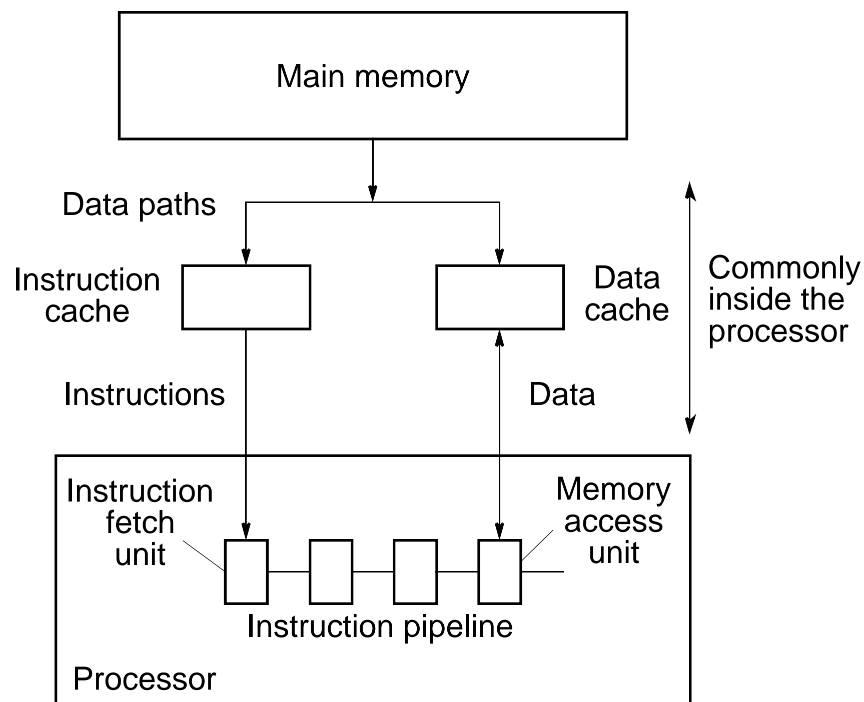
Barry Wilkinson 2000. All rights reserved.

Instruction and Data Caches

Several advantages if separate cache into two parts, one holding the data (a *data cache*) and one holding program instructions (an *instruction* or *code cache*):

- Write policy would only have to be applied to the data cache (assuming instructions are not modified.
- Designer may choose to have different sizes for the instruction cache and data cache, and have different internal organizations and line sizes for each cache.
- Separate paths could be provided from the processor to each cache, allowing simultaneous transfers to both the instruction cache and the data cache.

Particularly convenient in a pipeline processor, as different stages of the pipeline access each cache:



Barry Wilkinson 2000. All rights reserved.

Replacement policy

Policy to select a line to remove for an incoming line. (Not applicable for direct mapping which does not allow any choice).

Must be implemented in hardware, preferably such that selection can be made completely during main memory cycle for fetching new line.

Ideally, line replaced will not be needed again in the future. However, such future events cannot be known and decision has to made based upon facts that are known at the time. Classified as *usage-based* or *non-usage-based*.

A usage-based replacement algorithm for the fully associative cache needs to take the usage (references) to all stored lines into account.

A usage-based replacement algorithm for a set-associative cache needs to take only the lines in one set into account at replacement time.

Random replacement algorithm

Perhaps the easiest replacement algorithm to implement is a pseudo-random replacement algorithm.

A true random replacement algorithm would select a line to replace in a totally random order, with no regard to memory references or previous selections; practical random replacement algorithms can approximate this algorithm.

Barry Wilkinson 2000. All rights reserved.

First-in first-out replacement algorithm

Removes the line which has been in the cache for the longest time.

The first-in first-out algorithm would naturally be implemented with a first-in first-out queue of line addresses, but can be more easily implemented with counters. - not usually implemented.

Least recently used algorithm for a cache

The line which has not been referenced for the longest time is removed from the cache. Only those lines in the cache are considered.

The word "recently" comes about because the line is not the least used as this is likely to be back in memory. It is the least used of those lines in the cache, and all of these are likely to have been recently used otherwise they would not be in the cache.

LRU algorithm popular for cache systems and can be implemented fully when the number of lines involved is small. Several ways the algorithm can be implemented in hardware for a cache.

Barry Wilkinson 2000. All rights reserved.

Reference Matrix Method

The reference matrix method can be derived from the following definition (for 4 lines):

$B_5 = 1$ when line 3 is more recently used than line 2.

$B_4 = 1$ when line 3 is more recently used than line 1.

$B_3 = 1$ when line 3 is more recently used than line 0.

$B_2 = 1$ when line 2 is more recently used than line 1.

$B_1 = 1$ when line 2 is more recently used than line 0.

$B_0 = 1$ when line 1 is more recently used than line 0.

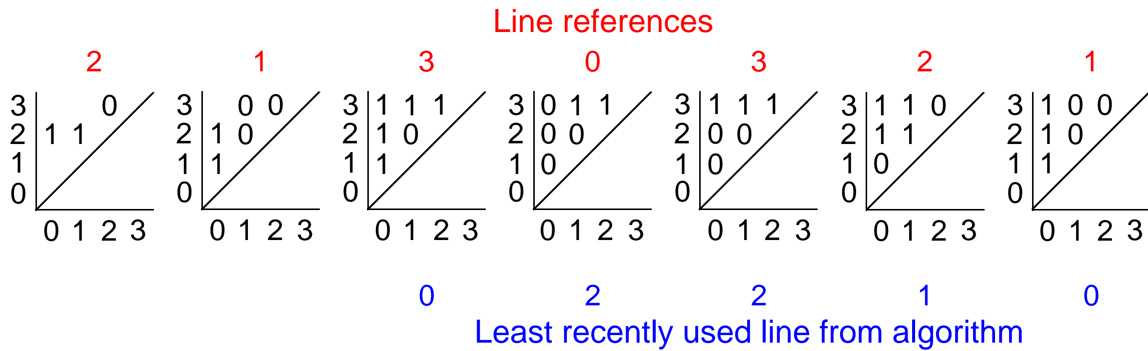
The bits B_5, B_4, B_3, B_2, B_1 and B_0 can be arranged as an upper triangular matrix of a $B \times B$ bits (B_0 the first row, B_1 and B_2 the second row, and B_3, B_4, B_5 the third row).

When the *i*th line is referenced, all the bits in the *i*th row of the matrix are set to 1 and then all the bits in the *i*th column are set to 0. (Prove)

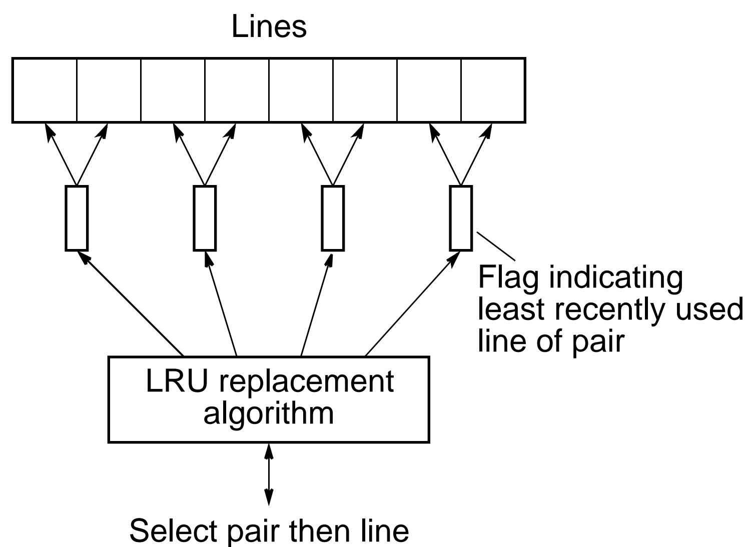
The least recently used line is one which has all 0's in its row and all 1's in its column, which can be detected easily by logic. (Prove)

Barry Wilkinson 2000. All rights reserved.

Least recently used replacement algorithm implementation using reference matrix

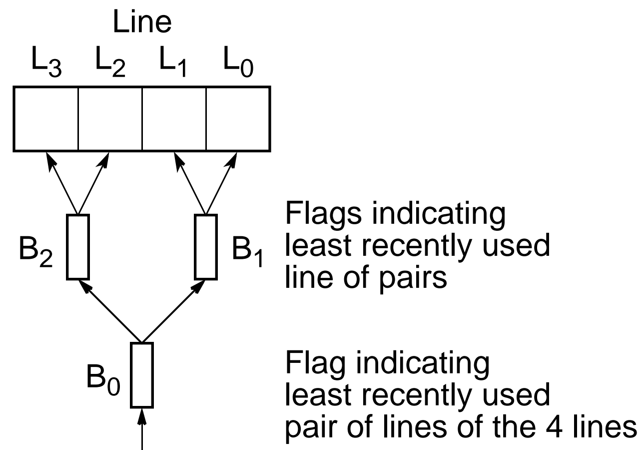


Two-stage replacement algorithm



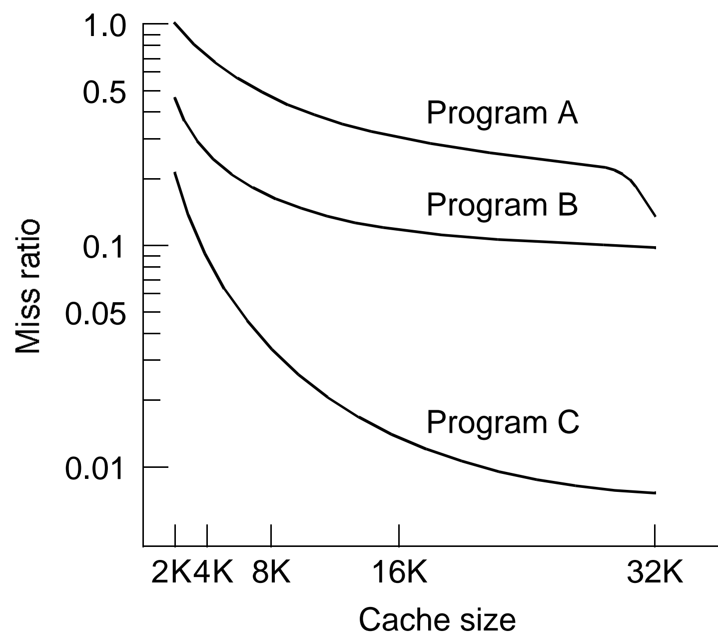
Barry Wilkinson 2000. All rights reserved.

Replacement algorithm using a tree selection



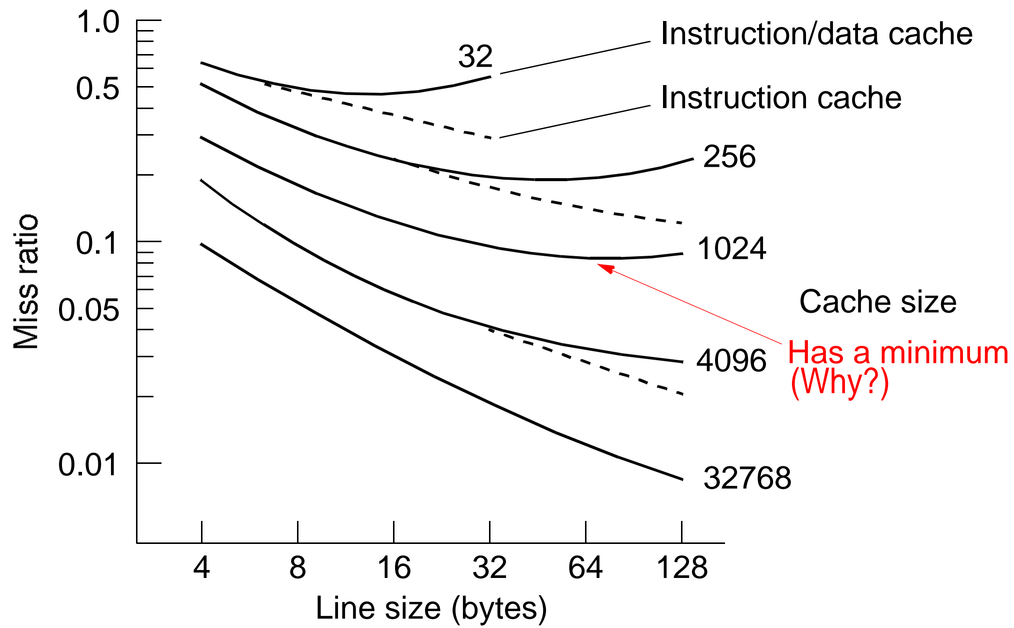
General Cache Performance Characteristics

Miss Ratio against Cache Size



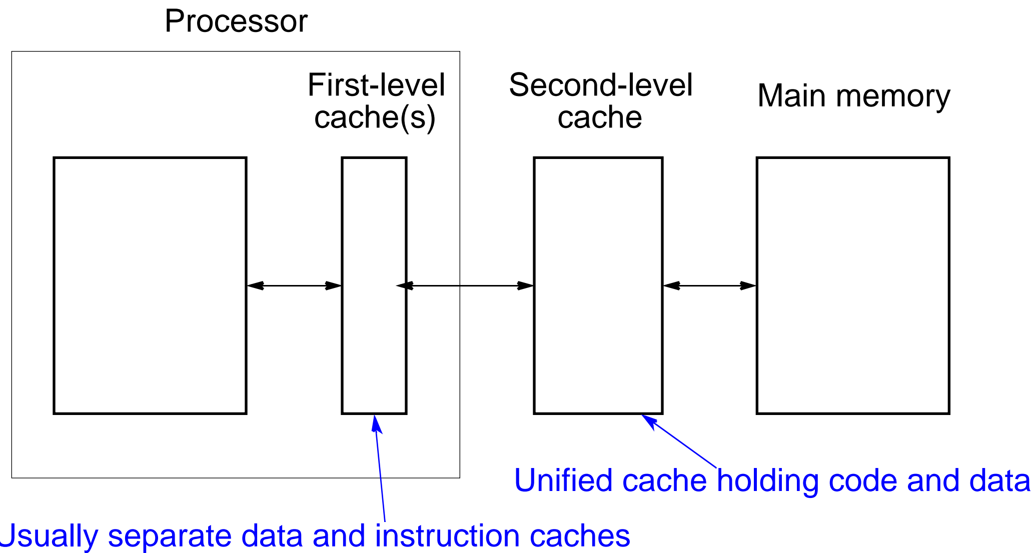
Barry Wilkinson 2000. All rights reserved.

Miss Ratio against Line Size



Second Level Caches

Most present-day systems use two levels of cache.



First-level cache access time matches processor. Second-level cache access time between main memory access time and first level cache access time:

Barry Wilkinson 2000. All rights reserved.

Second level cache equation

Can extend our previous equations to cover a second-level cache. Expanding t_c in:

$$t_a = t_c + (1 - h)t_m$$

we get:

$$t_a = [t_{c1} + (1 - h_1)t_{c2}] + (1 - h_2)t_m$$

where t_{c1} is the first-level cache access time, t_{c2} the second-level cache access time, t_m the main memory access time, h_1 is the first-level cache hit ratio, and h_2 is the combined first/second-level cache hit ratio, considering the two caches are one homogeneous cache system.

Most microprocessor families provide for second-level caches.

Memory Management

Methods of managing the memory hierarchy in a computer system.

Two separate issues for memory management:

1. Handling the main and disk memory hierarchy.
2. Providing memory protection.

Barry Wilkinson 2000. All rights reserved.

Paging/Virtual Memory

Objective - to make the main and secondary memories seem as though all the memory was all main random access memory. Based upon dividing memory space into pages that are transferred between the memories automatically.

The user is given the impression of a very large main memory space (*virtual memory*) which hides the actual memory space (the real memory space).

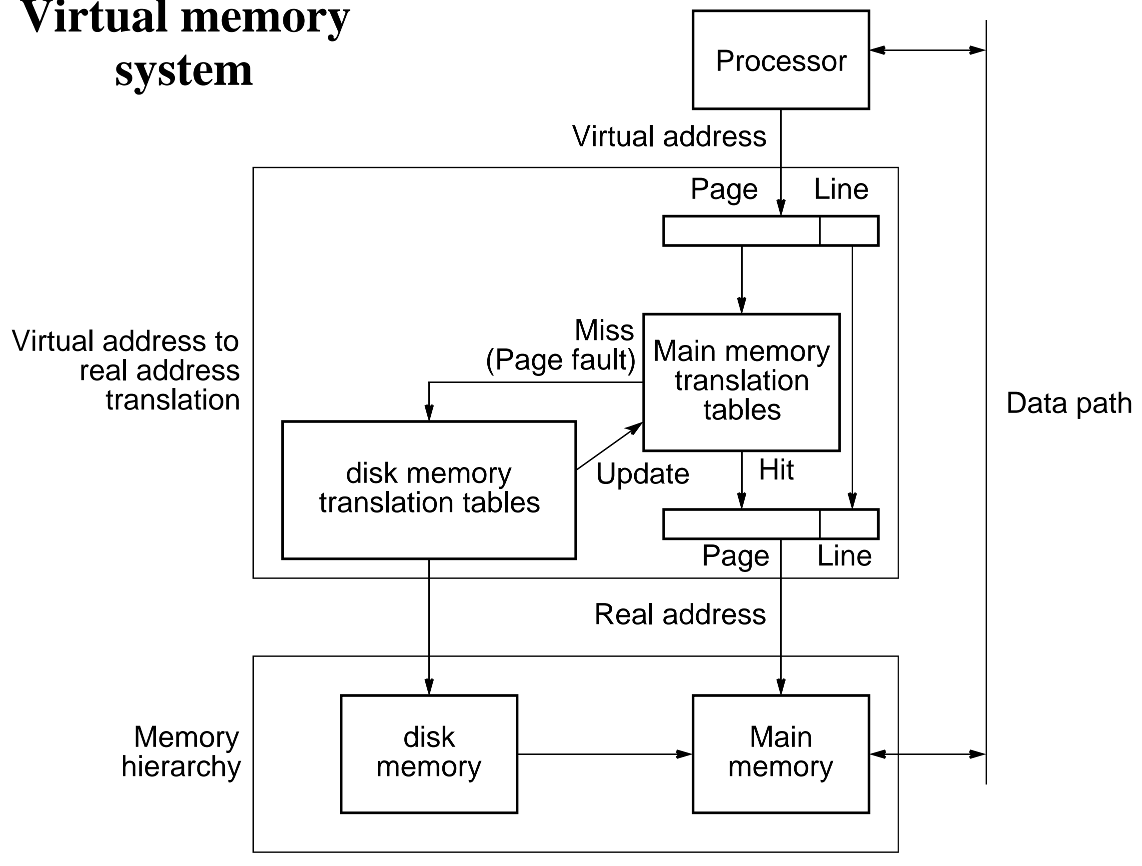
Separate addresses are used for the virtual memory space and the real memory space.

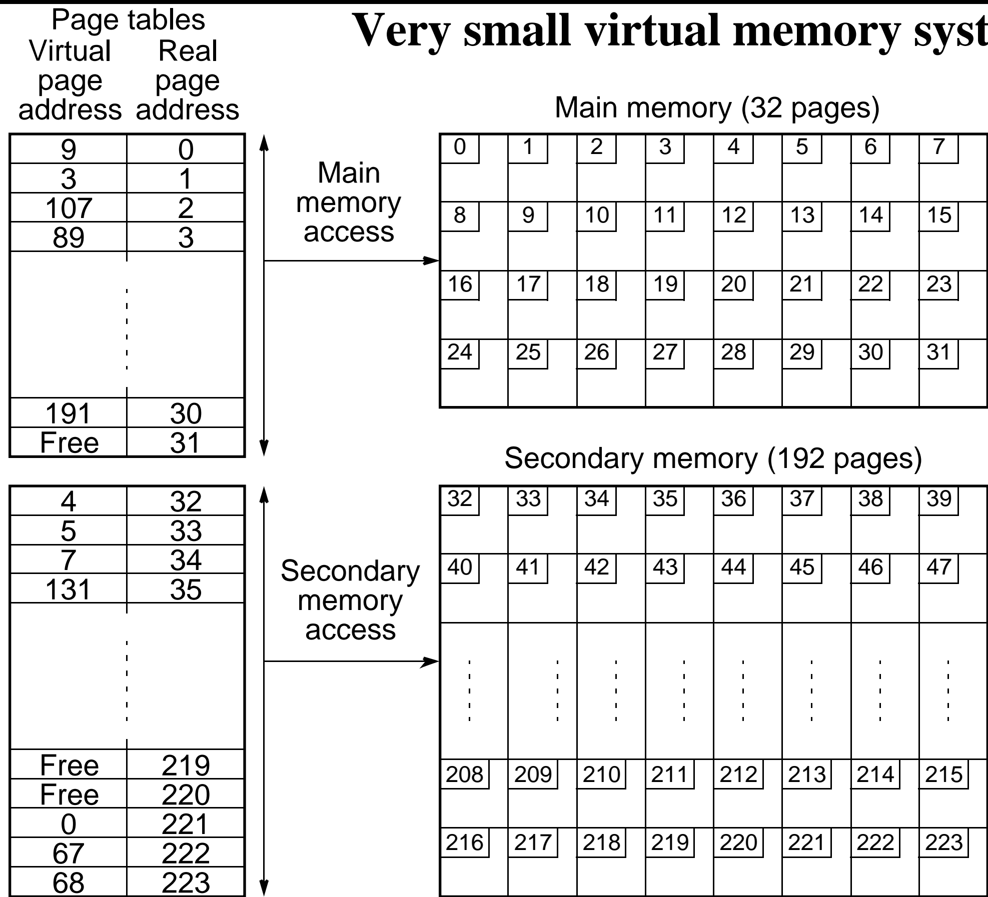
The actual memory addresses are called *real addresses*
The program generated addresses are called *virtual addresses*.

Real and virtual memory spaces both divided into blocks of words called *pages*.

Pages (selectable)size might be between 64 bytes and 4 Kbytes, depending upon design.

Barry Wilkinson 2000. All rights reserved.

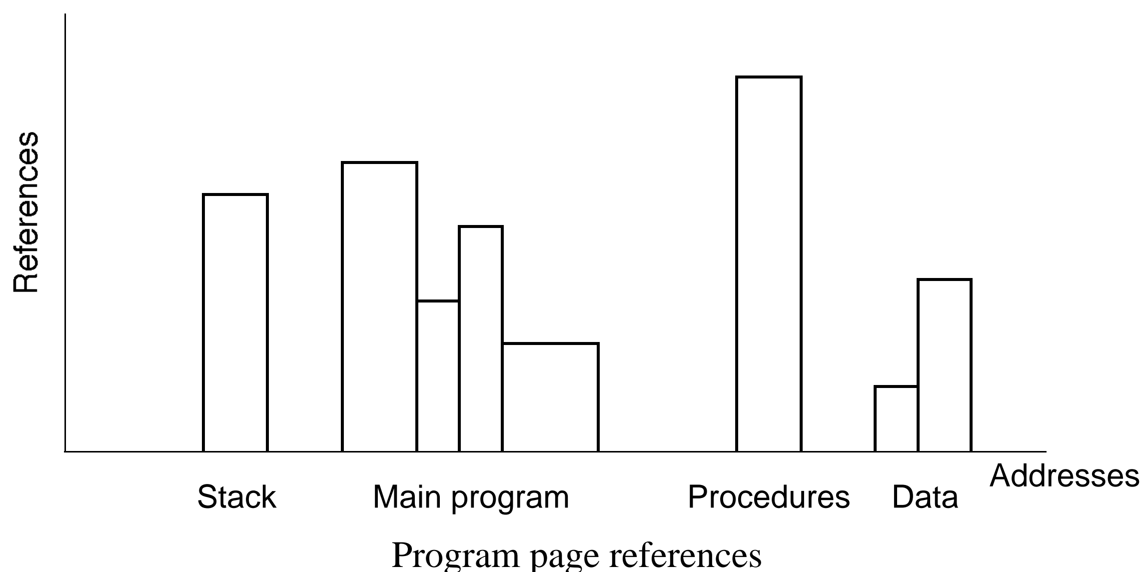




Virtual page address	Real page address
9	0
3	1
107	2
89	3
⋮	⋮
191	30
Free	31
4	32
5	33
7	34
131	35
⋮	⋮
Free	219
Free	220
0	221
67	222
68	223

Principle of Locality

Just as Principle of Locality causes caches to be successful, Principle of Locality makes paging successful. References are grouped into particular regions and many, if not all, locations are referenced several times.



Translation look-aside buffer (TLB)

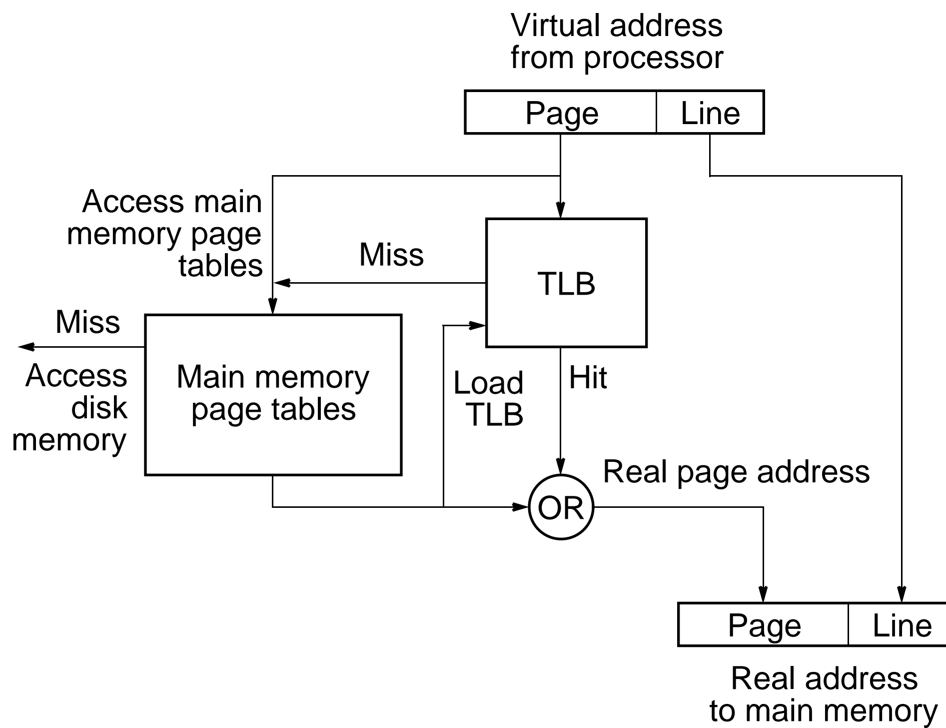
Number of pages in a modern computer system too large to hold all the main memory page table in a very high speed look-up table.

Given program characteristics embodied in the Principle of Locality, only those page addresses predicted as most likely to be used need be translated in hardware. The rest of the page references are initially handled by reading a main memory page look-up table.

The high speed page address translation memory holding the most likely referenced page entries is known as a *translation look-aside buffer* (TLB)

Barry Wilkinson 2000. All rights reserved.

Translation look-aside buffer



Data and Instruction TLBs

Since most processor have caches and separate data and instruction caches, it is reasonable to use separate TLBs for each type of reference.

A reference to data will use one TLB to translate its virtual address into a real address.

A reference to an instruction will use another TLB to translate its virtual address into a real address.

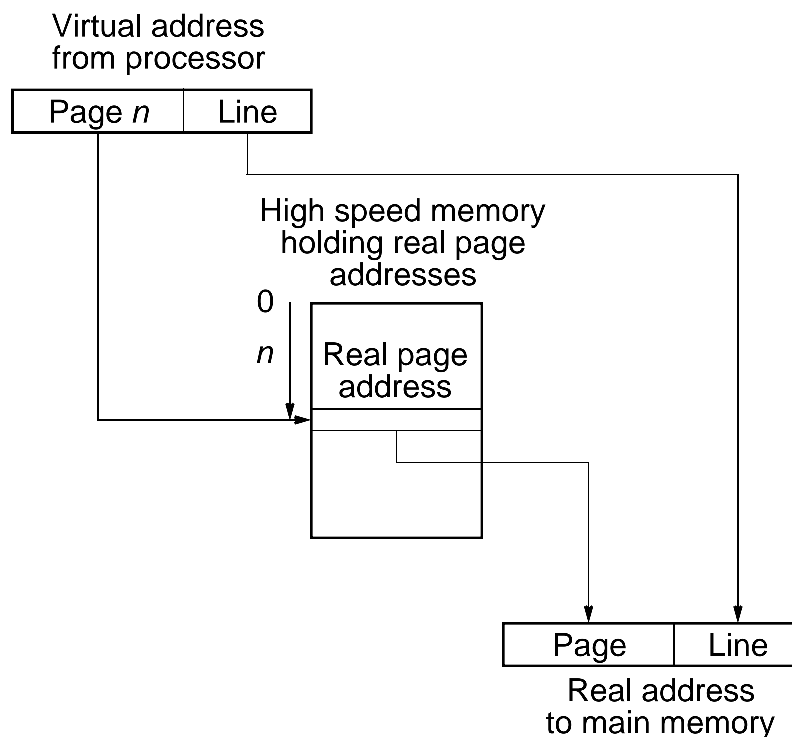
Barry Wilkinson 2000. All rights reserved.

Address translation

There are three basic hardware techniques to translate the virtual page address into a real page address:

1. (Pure) direct mapping.
2. Associative mapping.
3. Set-associative mapping.

(Pure) direct mapping address translation

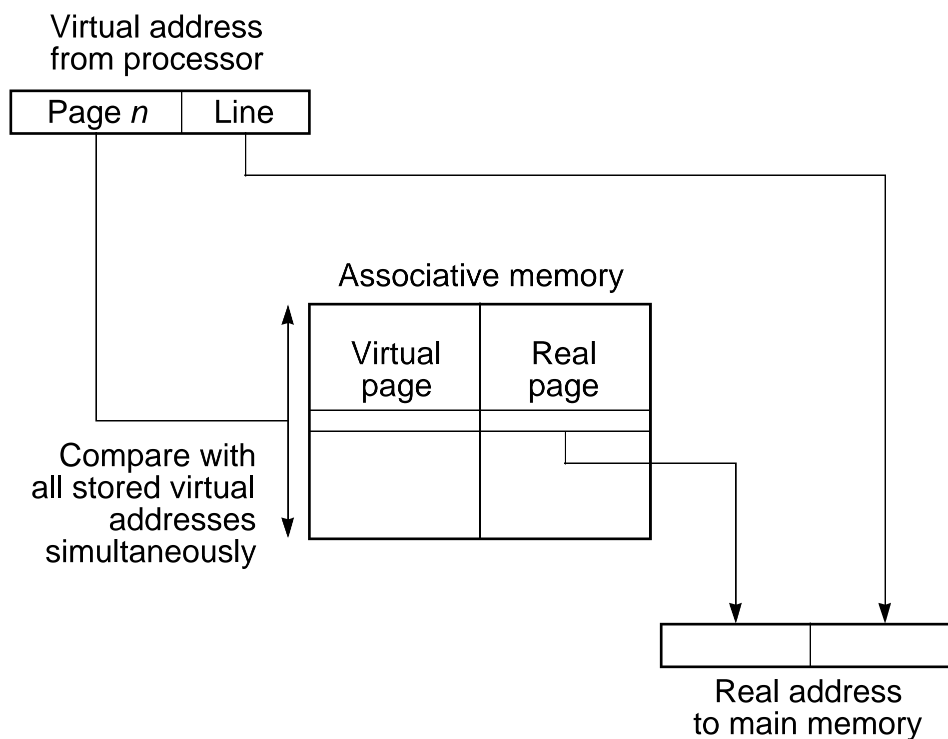


Barry Wilkinson 2000. All rights reserved.

(Pure) direct mapping technique shown not suitable for a TLB.

Direct method suitable for the main memory and second memory page tables, and will be the basis of these tables.

Fully associative mapping address translation

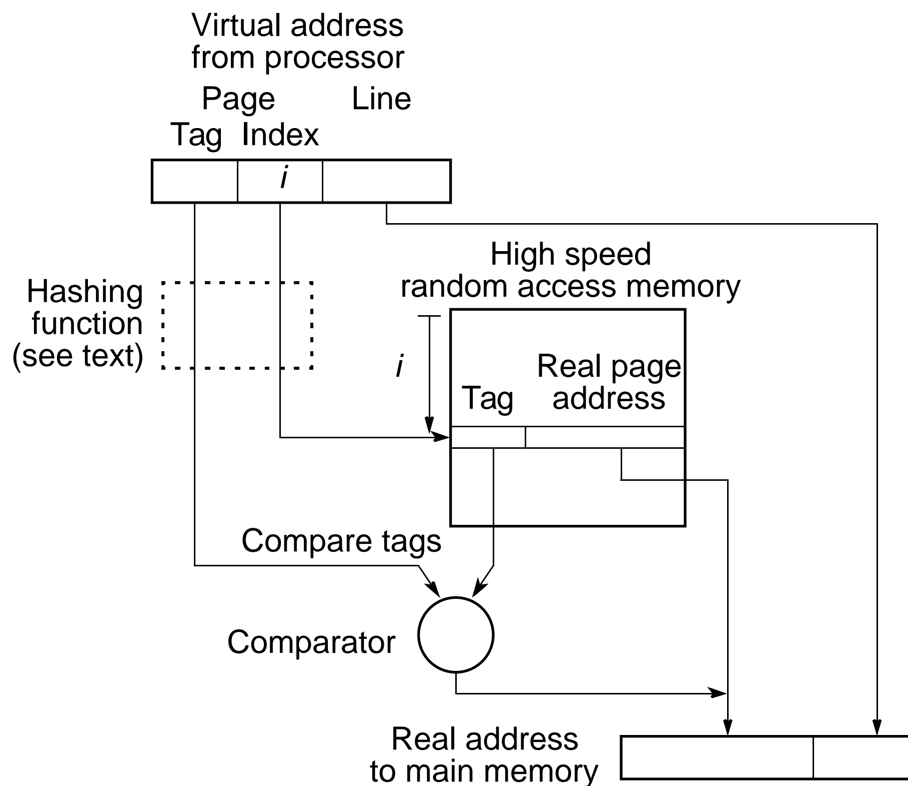


Barry Wilkinson 2000. All rights reserved.

Fully associative method is used in some TLBs.

However, large fully associative TLBs may be expensive to create and will operate slower than set-associative TLBs. Most TLBs are now set-associative. This does not come without a performance consideration as we shall see.

Set-associative mapping address translation (one-way)



Barry Wilkinson 2000. All rights reserved.

Many systems use two- or four-way set-associative TLBs.

Examples

Intel 486 has a 32-entry four-way set-associative TLB.

Motorola 68040 has two 64-entry four-way set-associative TLBs, one for the data cache and one for the instruction cache.

Pentium has two data TLBs, one 64 entry 4-way dual port TLB for 4 Kbyte pages and one 8 entry 4-way dual port TLB for 4 Mbyte pages and one instruction 32-entry 4-way TLB.

Hashing Functions

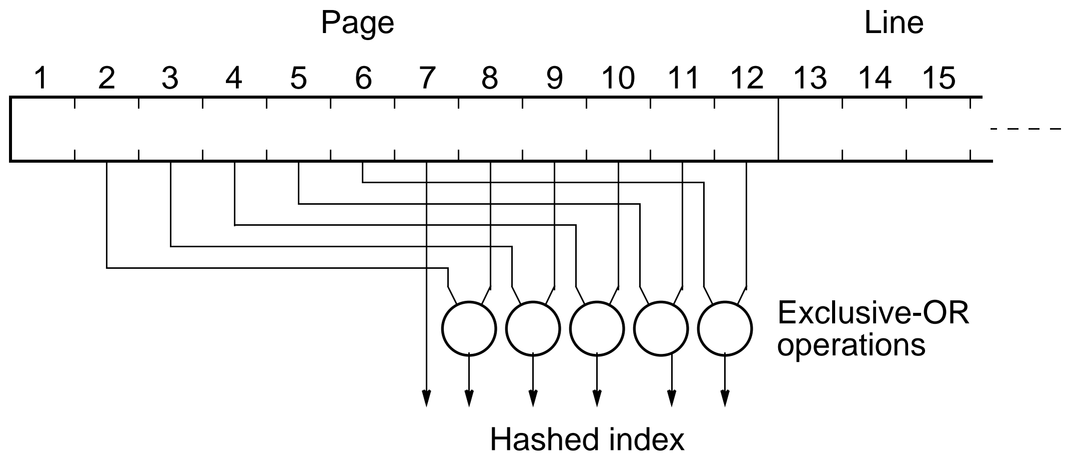
Set-associative TLB with index directly addressing the TLB has major disadvantage that only *n* pages with virtual addresses having the same lower page bits (index bits) can be translated with a set size of *n*.

The set size is often only one or two. The chance of virtual addresses having the same lower page bits is quite high.

To counteract this effect, alter bits accessing TLB, or "hash" bits to "randomize" the virtual page address before accessing the TLB.

Barry Wilkinson 2000. All rights reserved.

IBM 3033 page hashing function



Page size

Various page sizes used in paging schemes, from small pages of 64 bytes through to very large pages of 512 Kbytes. A common page size has been 4 Kbytes. Some systems provide for different page sizes for flexibility.

Example

Two page sizes can be selected in the Intel Pentium, either 4-Kbyte or 4-Mbyte pages. system software.

A small page of 64 bytes might be suitable for code while a larger page of 512 bytes might be suitable for data. A very large page size of say 4 Mbytes might suit graphics applications.

Barry Wilkinson 2000. All rights reserved.

Small page size

- Time taken in transferring a page between the main memory and the disk memory is short
- Large selection of pages from various programs can reside in main memory.
- Also reduces the storing of superfluous code which is never referenced.
- Necessitates a large page table
- *Table fragmentation* increases. This is the term used to describe the effect of memory being occupied by mapping tables and hence being unavailable for code/data.

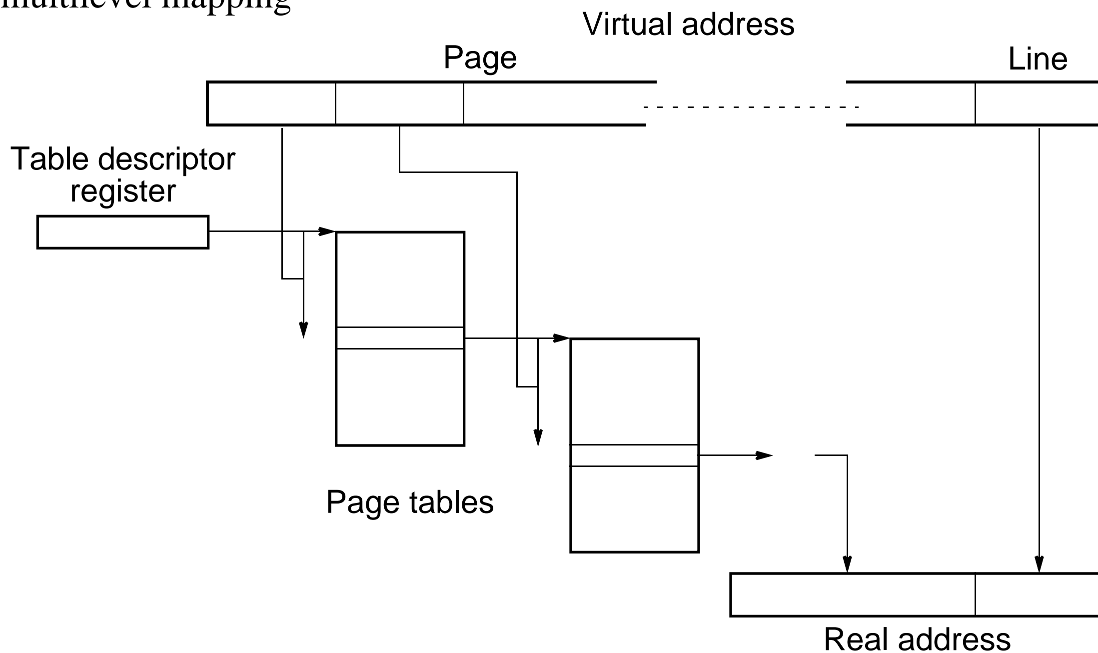
Large page size

- Requires a small page table but the transfer time is generally longer.
- Unused space at the end of each page is likely to increase – an effect known as *internal fragmentation* - on average, the last page of a program is likely to be 50 per cent full.

Barry Wilkinson 2000. All rights reserved.

Multilevel page mapping

Page table giving all virtual/real page associations for main memory requires considerable memory. To reduce this requirements, use two- or multilevel mapping



Handling page faults

A *page fault* occurs whenever page referenced is not already in main memory, i.e. when a valid page entry not found in TLB and main memory page tables.

When page fault occurs, the required page must be located in the disk memory using the disk memory page tables, and a page in the main memory must be identified for removal if there is no free space in the main memory.

Barry Wilkinson 2000. All rights reserved.

Three policies to consider when handling page faults:

1. Fetch policy – to determine when pages are loaded into the main memory.
2. Placement policy – to determine where pages are to be placed in main memory.
3. Replacement policy – to determine which page in the main memory to remove or overwrite.

Normal fetch policy - *demand paging* - wait until a page fault occurs and then loading the required page from the disk memory.

Placement policy - Original policy was to maintain one free page in the main memory for the incoming page. Another page is removed afterwards to create a free page for the next incoming page.

Barry Wilkinson 2000. All rights reserved.

Page replacement algorithms

Can be classified as:

1. Usage-based algorithms.
2. Non-usage-based algorithms.

In a usage-based algorithm the choice of page to replace is dependent upon how many times each page in the main memory has been referenced. Non-usage-based algorithms use some other criteria for replacement.

Implementing usage-based algorithms

Hardware is necessary to record when pages are referenced.

Use (or *accessed*) bit with each page entry - set if corresponding page referenced and automatically reset when the bit is read. Use bits are read under program control.

Use bits usually scanned at perhaps after 1 ms of process time to obtain an approximation of the usage.

Barry Wilkinson 2000. All rights reserved.

Page table has other bits to assist the replacement algorithm:

Modified (or *written*, *changed* or *dirty*) bit. - set if write operation performed on any location within page. Not necessary to write an unaltered page back to disk memory if a copy maintained there.

Very occasionally, unused bit set to 1 when the page loaded into main memory and reset to 0 when subsequently referenced. May be helpful to ensure that page demanded not removed before being used.

Protection bits - concerned with controlling access to pages.

Random replacement algorithm

Page is chosen randomly at page fault time; there is no relationship between the pages or their use. Does not take the principle of locality of programs into account. Simple to implement but is not widely applied to TLBs.

Examples

VAX 11/780 translation buffer (TLB), the TLB in the Intel i860 RISC processor.

Barry Wilkinson 2000. All rights reserved.

Least recently used replacement algorithm

Page which has not been referenced for the longest time is transferred out at page fault time.

Poses practical problems for a true implementation since there are many pages to consider.

A common approximation - at intervals, say after every 1 ms, all of use bits are examined by the operating system and automatically reset when read.

A record of the number of times the bits are found set to 1 would give an approximation of the usage in units of the interval selected.

Approximation becomes closer to a true LRU algorithm as the interval is decreased.

TLB performance

If the page address is not found in the TLB, a TLB miss occurs, and a significant overhead occurs in searching the main memory page tables, even when the page is already in the main memory.

The TLB like a data cache. Basic cache equations also apply to TLB, i.e. the address translation time, t_t, is given by:

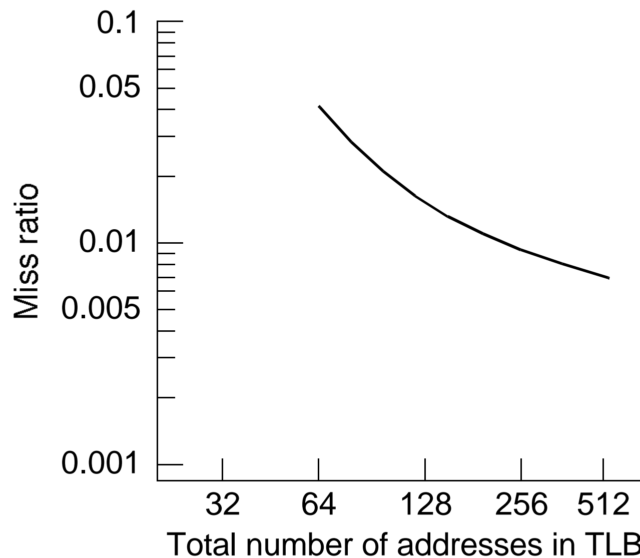
$$t_t = t_{tlb} + (1 - h_{tlb})t_{mt}$$

where t_{tlb} is the translation time of the TLB (hit or miss) and t_{mt} is the translation time looking in main memory tables on a TLB miss.

The TLB miss ratio is given by $(1 - h_{tlb})$. Typically the TLB miss ratio (miss rate) is very low indeed, perhaps less than 0.05 per cent.

Barry Wilkinson 2000. All rights reserved.

TLB miss ratio against size



Virtual memory systems with cache memory

Can insert cache after TLB virtual/real address translation, so that the cache holds real address tags and the comparison of addresses is done with real addresses.

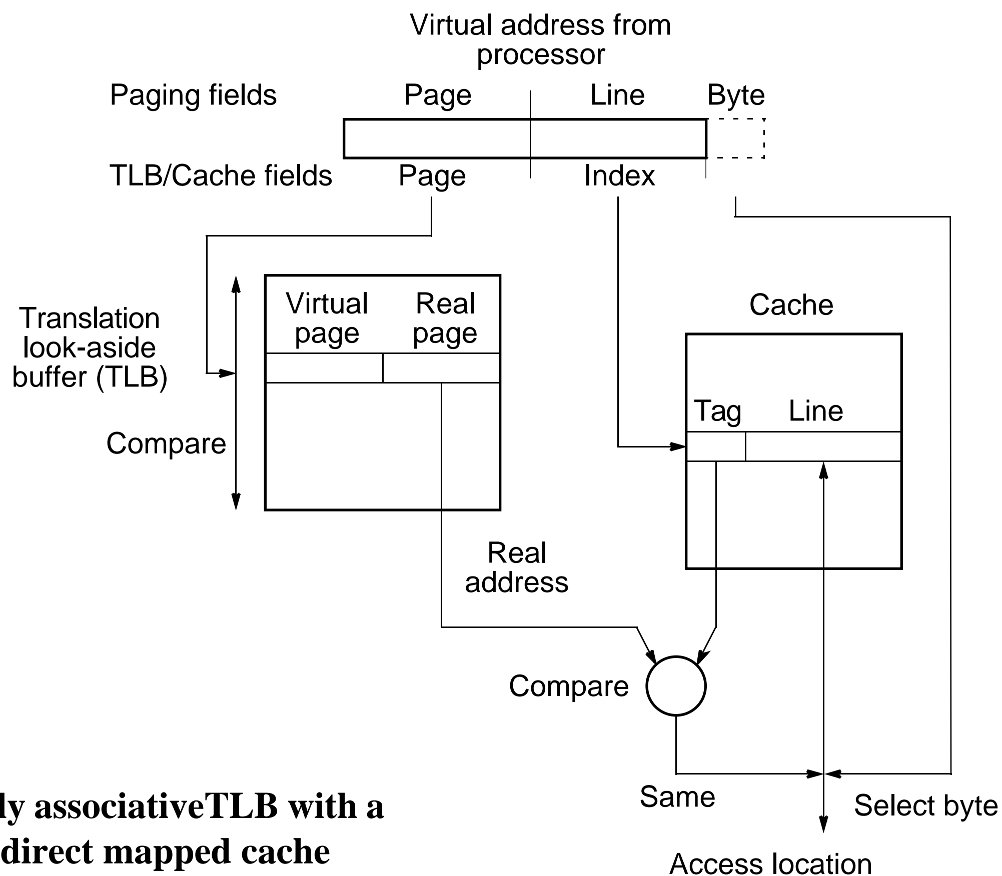
Alternatively, can insert cache before TLB virtual-real translation so that the cache holds virtual address tags and the comparison of addresses is done using virtual addresses.

Former case, which is much less complicated and has fewer repercussions on the rest of the system design.

Barry Wilkinson 2000. All rights reserved.

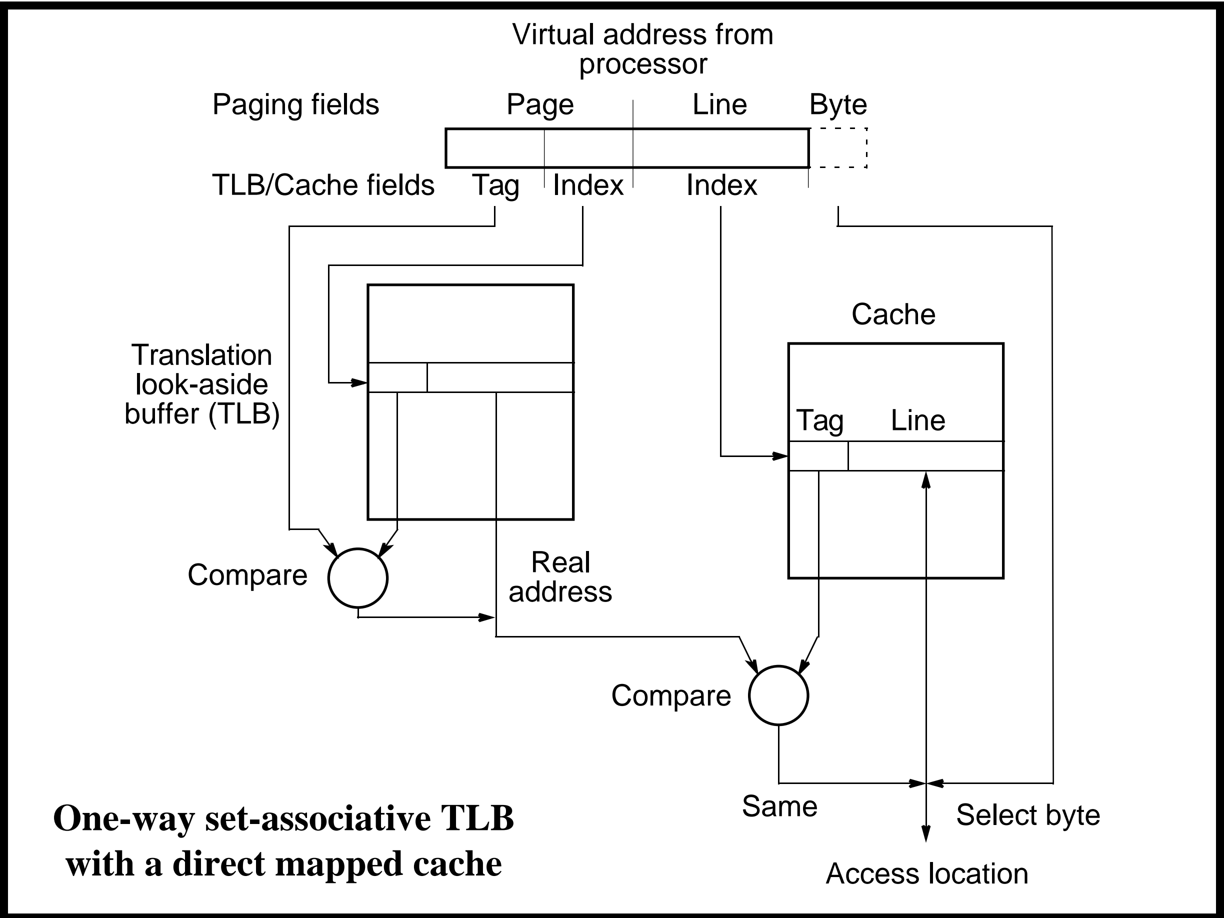
Addressing cache with real addresses

Common to perform the TLB virtual-real translation at the same time as some independent part of the cache selection operation to gain an improvement in speed.

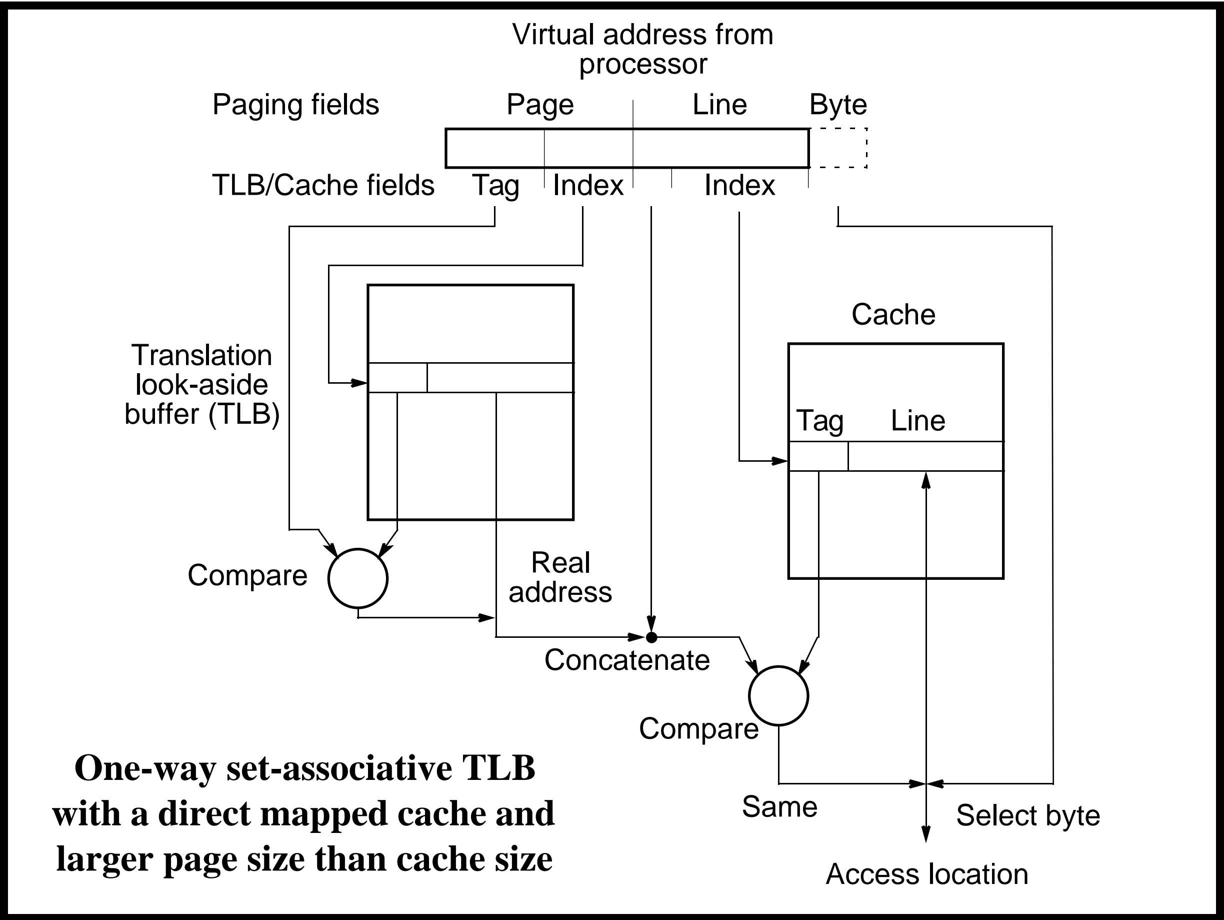


Fully associativeTLB with a direct mapped cache

Barry Wilkinson 2000. All rights reserved.



One-way set-associative TLB with a direct mapped cache



One-way set-associative TLB with a direct mapped cache and larger page size than cache size

Barry Wilkinson 2000. All rights reserved.

Addressing cache with virtual addresses

If the cache is addressed with virtual addresses, these addresses are immediately available for selecting a word within the cache. Only on a cache miss would it be necessary to translate a virtual address into a real address, and there is more time then.

Potential increase in speed over a real addressed cache.

Complications

Possible for different virtual addresses in different processes to map into same real address. Such virtual addresses known as *synonyms* – from the word denoting the same thing(s) as another but suitable for different contexts.

Synonyms occur:

- If the addressed location is shared between processes
- If programs request the operating system to use different virtual addresses for the same real address.
- When an input/output device uses real addresses to access main memory accessible by the programs.
- Can also occur in multiprocessor systems when processors share memory using different virtual addresses.

It is also possible for the same virtual address generated in different processes to map into different real addresses.

Barry Wilkinson 2000. All rights reserved.

Handling Synonyms

- Process or other tags could be attached to the addresses to differentiate between virtual addresses of processes,
- Synonyms could be disallowed by placing restrictions on virtual addresses.
- Could be handled by use of a *reverse translation buffer* (RTB). On a cache miss, the virtual address is translated into a real address using the virtual-real translation look-aside buffer (TLB) to access the main memory. When the real address has been formed, a reverse translation occurs to identify all virtual addresses given under same real address.

Real addressed cache access time

(At least) six combinations of accesses:

1. Address in the translation look-aside buffer, data in the cache.
2. Address in the translation look-aside buffer, data in the main memory.
3. Address in the cache, data in the cache.
4. Address in the cache, data in the main memory.
5. Address in the main memory, data in the cache.
6. Address in the main memory, data in the main memory.

Barry Wilkinson 2000. All rights reserved.

Example

Suppose no overlap between translation look-aside buffer translation and cache access (a rather unlikely situation) and the following times apply:

Translation look-aside buffer address translation time (or to generate a TLB miss)	= 25 ns
Cache time to determine whether address in cache	= 25 ns
Cache data fetch if address in cache	= 25 ns
Main memory read access time	= 200 ns
Translation look-aside buffer hit ratio	= 0.9
Cache hit ratio	= 0.95

Access times and probabilities of the various access combinations are:
:

Access times and probabilities of the various access combinations

Access time		Probabilities	
25 + 25 + 25	= 75 ns	0.9 × 0.95	= 0.855
25 + 25 + 200	= 250 ns	0.9 × 0.05	= 0.045
25 + 25 + 25 + 25 + 25	= 125 ns	0.1 × 0.95 × 0.95	= 0.09025
25 + 25 + 25 + 25 + 200	= 300 ns	0.1 × 0.95 × 0.05	= 0.00475
25 + 25 + 200 + 25 + 25	= 300 ns	0.1 × 0.05 × 0.95	= 0.00475
25 + 25 + 200 + 25 + 200	= 475 ns	0.1 × 0.05 × 0.05	= 0.00025

Average access time is given by:
$(75 \times 0.855) + (250 \times 0.045) + (125 \times 0.09025) + (300 \times 0.00475) + (300 \times 0.00475) + (475 \times 0.00025) = 89.63$ ns

Barry Wilkinson 2000. All rights reserved.

Segmentation

Purpose of segmentation is to organize the programs in memory so that the operating system can relocate programs in the main and disk memory easily, and to provide protection from unauthorized access/execution.

Although the way this is done looks similar to virtual memory/paging, not the same purpose as paging which has a hardware motive, to manage the memory hierarchy in an automatic way.

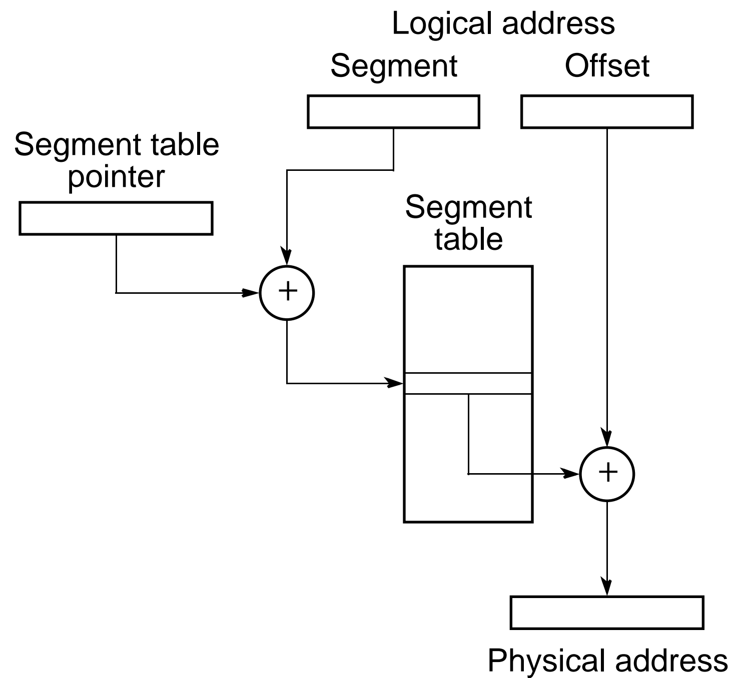
A *segment* is a block of contiguous locations. Segments may be of different sizes since programs are of different sizes.

Each address generated by the processor -*logical address* - is composed of a *segment number* and a *displacement* within the segment.

Segment number is translated into the start of the segment in memory and offset added to form the *physical address* .

Barry Wilkinson 2000. All rights reserved.

Segmentation address translation



Important aspect

The segment number and offset are separate entities and any alteration to the offset by the program cannot affect the segment number.

Once the maximum offset is reached (assuming that the segment grows with increasing addresses) adding one to the offset should create an error condition.

Barry Wilkinson 2000. All rights reserved.

Segment table

Incorporates additional information, including:

1. Segment length.
2. Memory protection bits.
3. Bits for the replacement algorithm.

Segment length

Length of each segment stored to prevent programs referencing a location beyond the end of a particular segment. If the offset in the virtual address is greater than the stored length (limit) field, an error signal is generated.

Barry Wilkinson 2000. All rights reserved.

Memory Protection

Memory protection involves preventing specified types of access to the addressed location and discarding or stopping the address translation occurring. The protection applies to all of the locations in the segment and not to particular locations. Typically, by setting bits in the segment tables, any segment can be assigned as allowing:

1. Read access
2. Write access
3. Execute access

for user, group, and system (e.g. UNIX).

Replacement algorithm

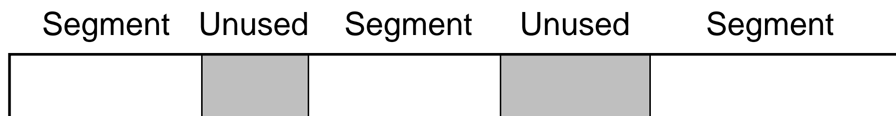
Can be similar to the replacement algorithm in a paged system except that it needs to take the varying size of the segments into account when allocating space for new segments.

Use flag usually sufficient to implement a replacement algorithm or approximations to a replacement algorithm.

Barry Wilkinson 2000. All rights reserved.

Placement algorithm

The variable size of segments causes some additional problems in main memory allocation. During operation, with segments returned to the disk memory, the main memory will become a "checkerboard":



External fragmentation. - leaving small spaces which cannot be used subsequently.

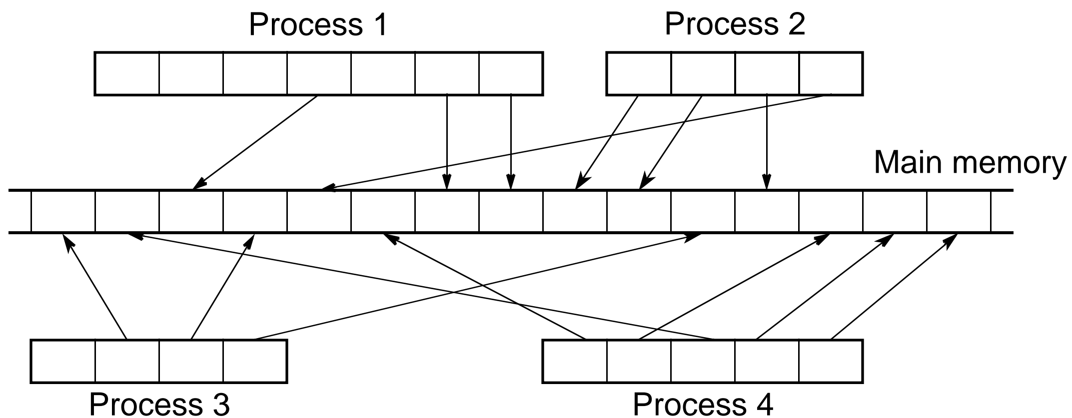
Placement algorithms

- first fit
- best fit
- worst fit.

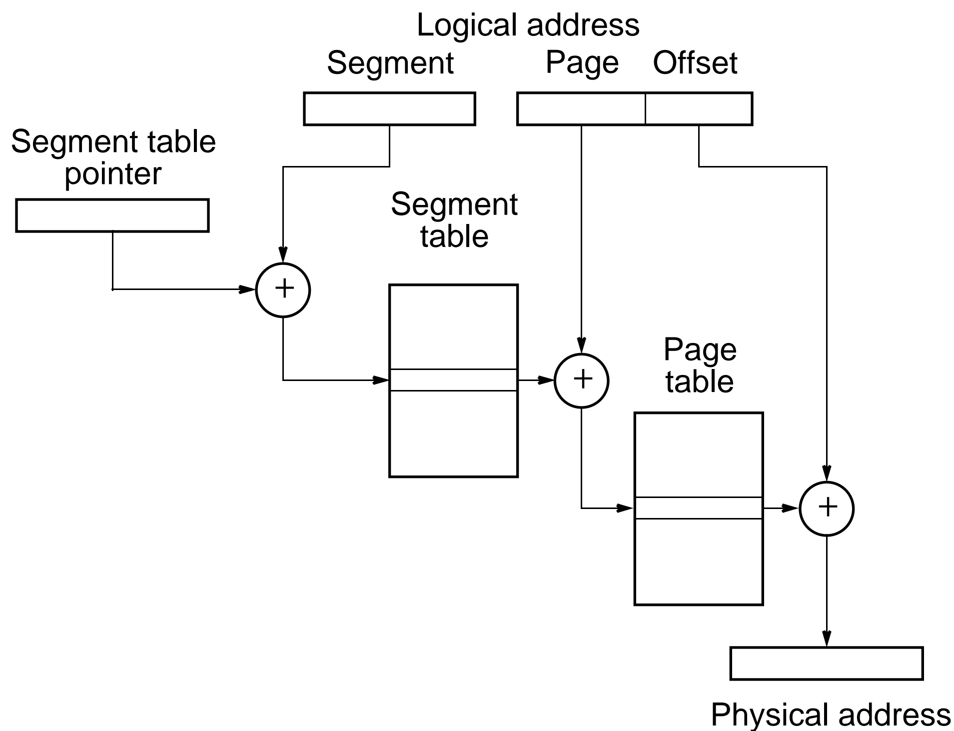
Barry Wilkinson 2000. All rights reserved.

Paged segmentation

Segmentation and paging are usually combined, to gain the advantages of both systems, i.e. the logical structure of segmentation and the hardware mapping between main and disk memory of paging.

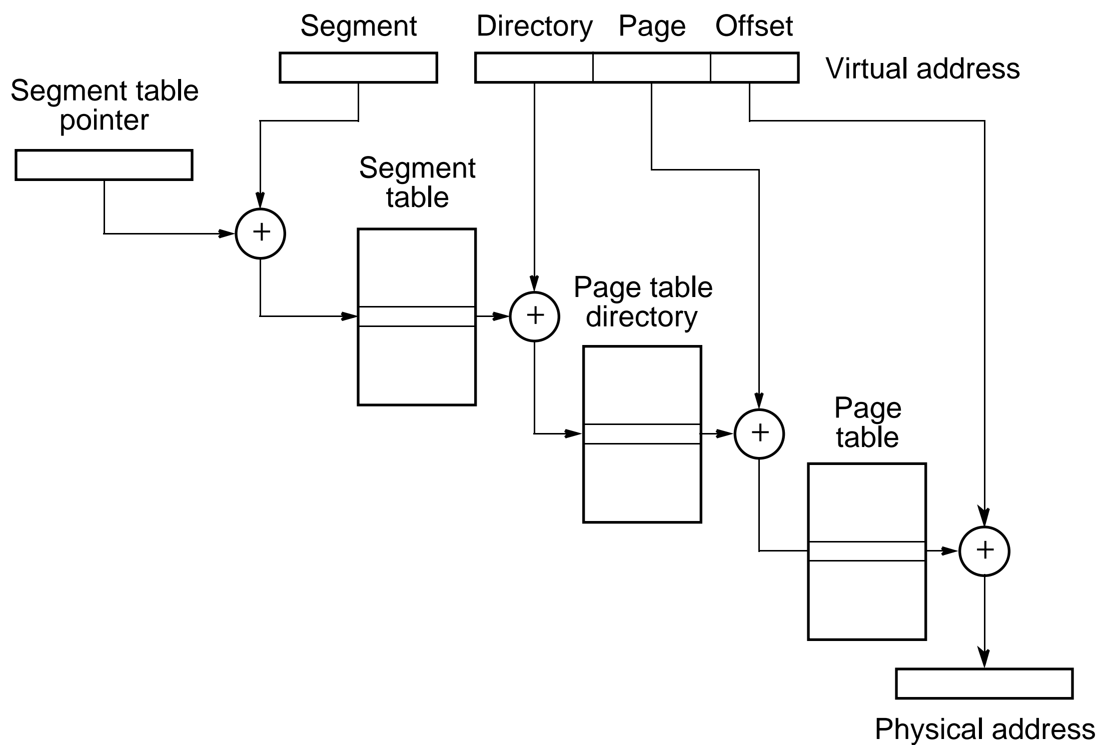


Paged segmentation address translation



Barry Wilkinson 2000. All rights reserved.

Two-level paging with segmentation



Barry Wilkinson 2000. All rights reserved.