

Memory Design

Processor - Memory Interface

The memory that the processor accesses must be **random access** memory - memory in which *individual* memory locations can be accessed in any order at the same high speed.

The memory should operate at a very high speed, preferably at a speed that **matches** the processor, so as not to slow the system down.

Access time - time to select memory location and obtain stored information or write new information.

Main Semiconductor Memory

Main semiconductor memory uses **dynamic memory cells*** to obtain high capacity and low cost.

Originally **asynchronous dynamic random access memory** (DRAM) - operates with access times of **50-70 ns**.

“Asynchronous” - memory operation is not synchronized to the processor clock - processor makes a request for a memory location and has to wait whatever time it takes to access the location.

* see ITCS 3182 notes

Processor-Memory Bottleneck

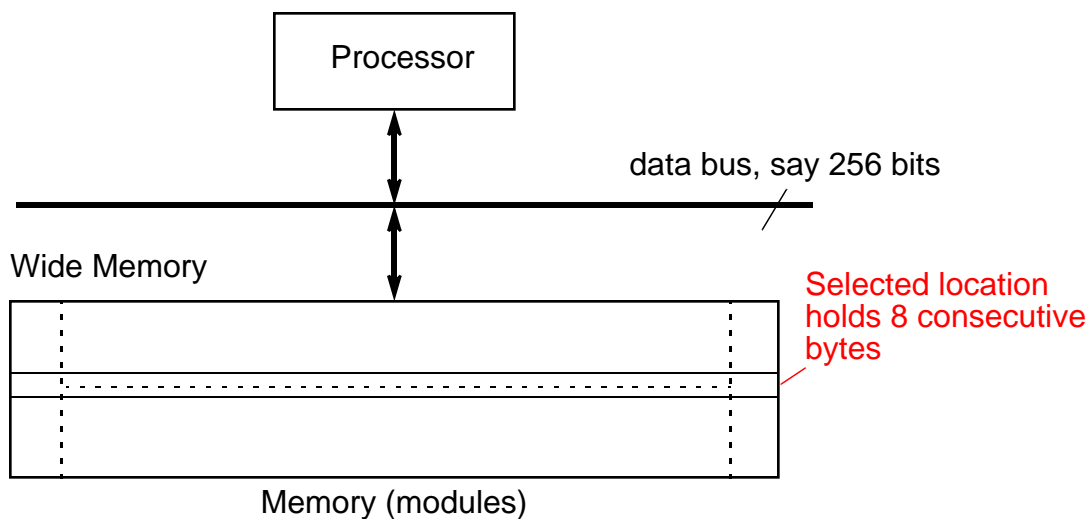
Whereas processor clock speeds have increased significantly over the years, now exceeding 1 GHz (1ns), **access time of basic dynamic memory cell** has not increased significantly*.

Various **architectural techniques** used to overcome this limitation.

* original DRAM in 1970's access times in order 100-150 ns but now 50-70 ns.

Increasing the Memory bandwidth

Memory **bandwidth** - number of bits transferred in a second.
 Bandwidth can be increased by simply increasing number of bits stored in locations and providing more wires to/from memory, e.g.:



Essentially accessing groups of sequential locations simultaneously

Disadvantages

- Cost of wires (big bus)
- Processor may not always requires sequential locations
- Limits in what increments memory can be expanded.

Page Mode

Used in **EDO (Extended Data Output) DRAM**.

Utilizes the internal row/column timing of DRAM.

First select a row of memory cells.

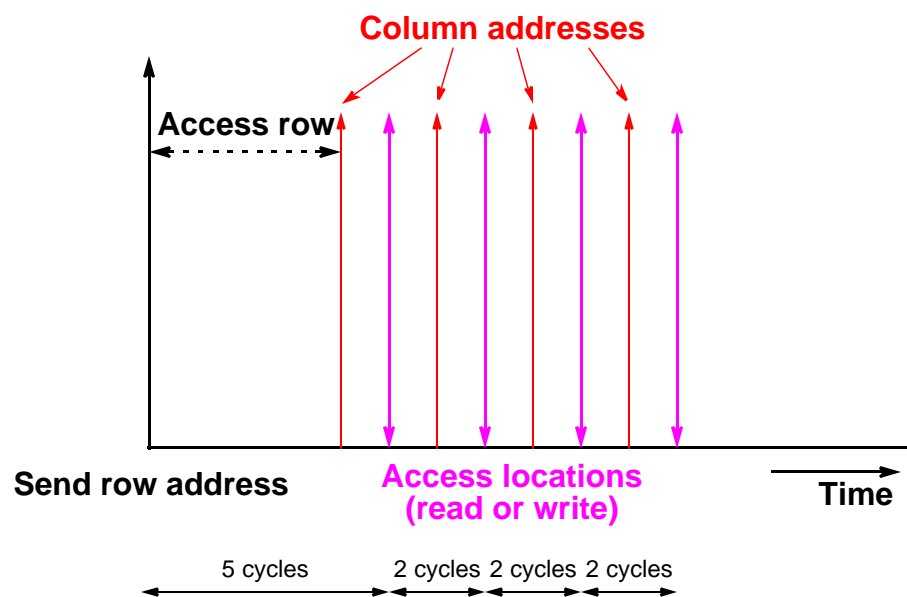
Then select series of columns one after the other to access cells on row in quick succession.

Takes normal access time (60 ns) to access first location but subsequent ones faster.

Example

5-2-2-2 timing - 5 cycles for first location, 2 cycles for each 3 subsequent locations.

Representative EDO DRAM Timing



Disadvantage

Subsequent locations requested by processor may not on same row.

Synchronous DRAM (SDRAM)

Introduced in the last few years replacing the original asynchronous DRAM but still using same basic dynamic memory cell design.

Access time of memory cell essentially the same.

Speed improvement obtained by synchronizing memory operation with processor and a **burst mode** in which a group of locations accessed one after the other at high speed after an initial read/write latency.

Example

5-1-1-1 timing - 5 cycles to obtain the first location, next 2 each after one cycle (4 accesses in 8 cycles).

Synchronous DRAM (SDRAM) Performance

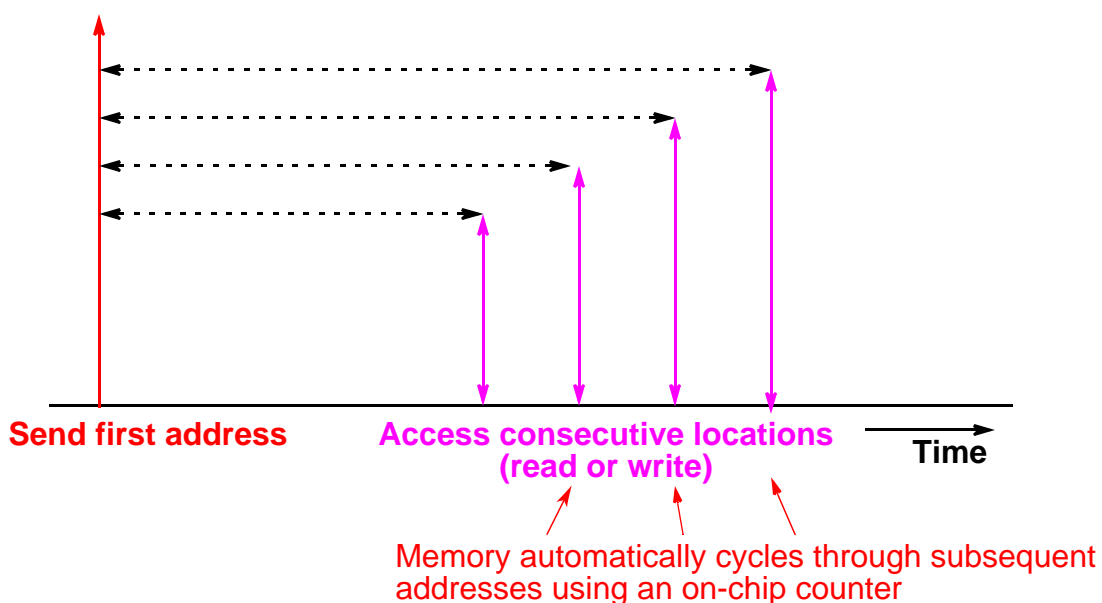
Performance now measured in clock frequency as well as in nanoseconds.

Example

100 Mhz (or 10 ns) SDRAM

This does not mean that the access time is 10 ns - in burst mode, a group of consequent locations can accessed at 10 ns intervals.

Representative Burst Mode Timing

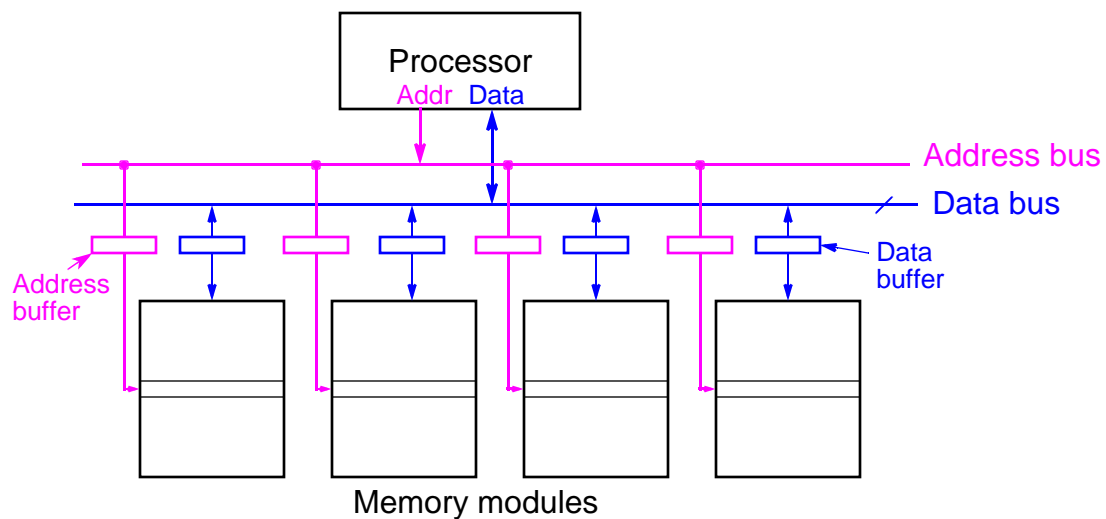


Disadvantage

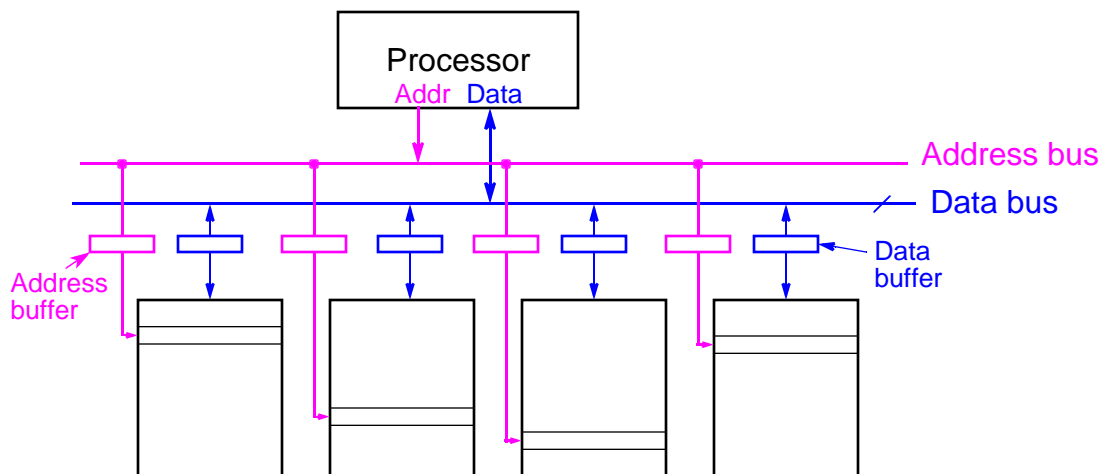
Subsequent locations requested by processor may not be sequential.

Interleaved Memory

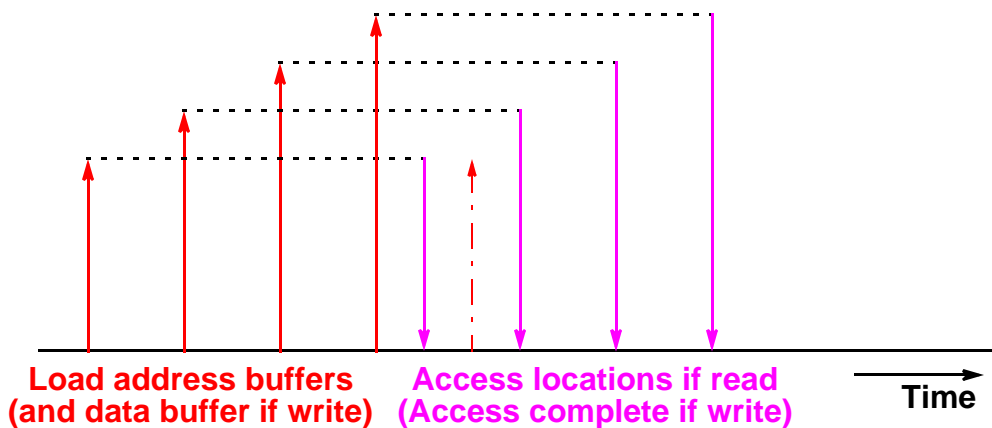
A general memory technique. Memory modules can be accessed at same time but accesses to each memory module “interleaved.”



If have separate address buffer for each memory module, possible for different locations within each memory module accessed, i.e.

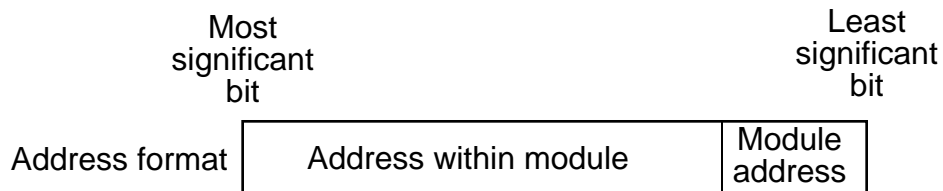


Representative Interleaved Memory Timing

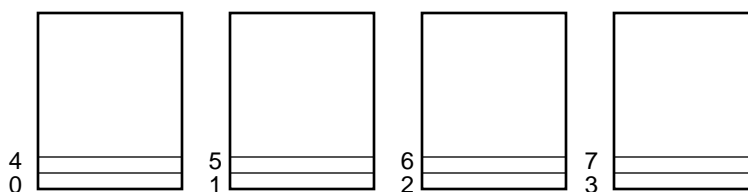


Low Order Interleaving

Lower bits of address select memory module and upper select location within module - consecutive locations in different memory locations:



Example: 4 memory modules



Advantages

Works well if accessing series of locations each in consecutive memory modules.

Disadvantages

Cannot handle arbitrary request patterns (even with separate address buffers) - notably sequence of locations within same memory module.

While burst mode, memory interleaving, and even increasing the number of wires in the memory bus, are ways of trying to match the speed of processor with the memory and are used, generally in addition, various levels of memory are provided with only that very close to the processor operating at processor speeds - the levels are called a memory hierarchy.

Memory Hierarchy

A computer system has a number of levels of memory for **economic reasons** and to provide some **non-volatile storage**.

The higher the speed of memory the more expensive it is. High speed semiconductor random access memory is also volatile, and information needs to be backed up on non-volatile memory.

The memory attached directly to the processor should operate at a speed that matches speed of memory requests from processor.

Lecture 14

Memory Hierarchy

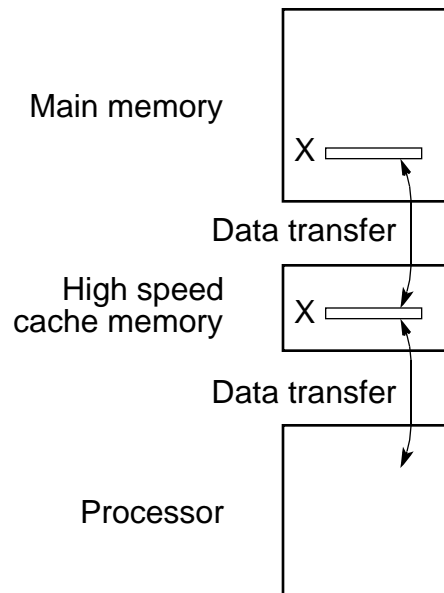
	Access time (representative)
Processor registers	1 ns (1GHz)
Cache memory (usually two levels)	
L1 cache	5 ns
L2 cache	15 ns
Main memory	60 ns (basic access time)
Disk memory	10 ms
CD drives	
Tape backup	10 mins

Cache Memory

Processor operates much faster than the main memory can.

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

Cache Memory



If same instructions 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, i.e. the access time would be:

$$t_a = t_m + t_c$$

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

Fortunately, 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.)

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

Temporal locality is essential for an effective cache.

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. Also seen in data accesses, as data items are often stored in sequential locations.

Spatial locality **helpful** in the design of a cache but **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.

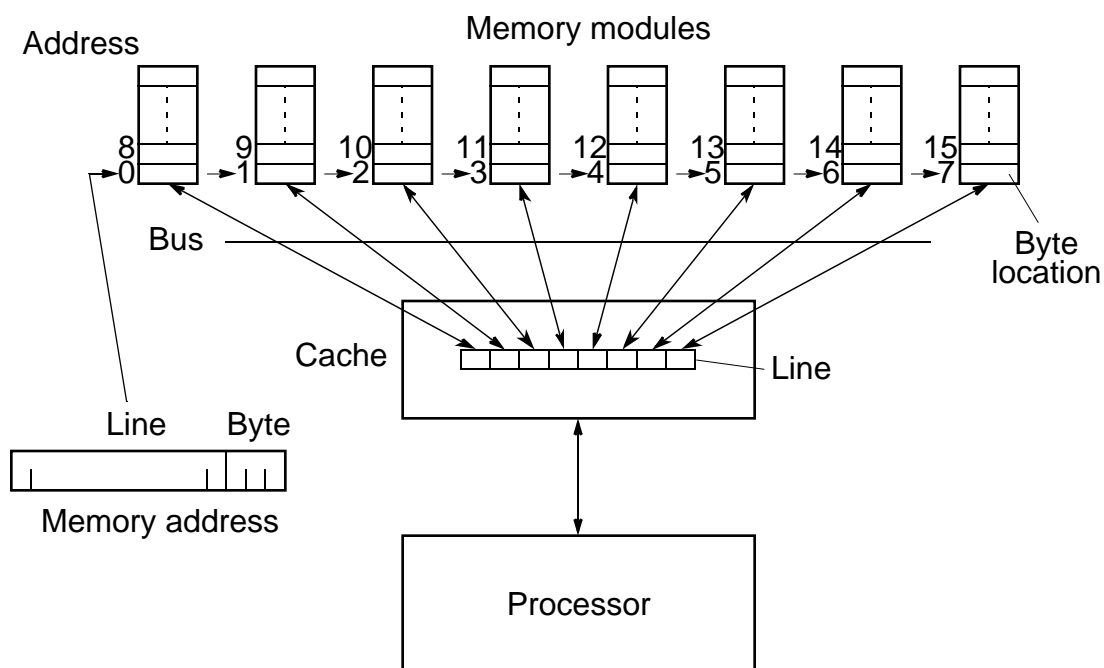
Example

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, 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.



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

Let number of memory modules and line size be m (words).

It takes t_m seconds to access main memory and load a cache line or an average of t_m/m for each word. (For a perfect match $t_m/m = t_c$)

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

$$\text{Average access time} = \frac{t_m/m + nt_c}{n}$$

assuming must load cache before accessing words.

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:

$$\text{Average access time} = \frac{(25 + 10 \times 5)}{10} = 7.5 \text{ ns}$$

The average access time is approximately t_c for large n .

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 first access must be to the cache before an access made to main memory. Only read requests 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 $5 + 0.15 \times 50 = 12.5$ 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.

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.

Average memory access time = hit time + miss rate \times 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_pen} for t_m

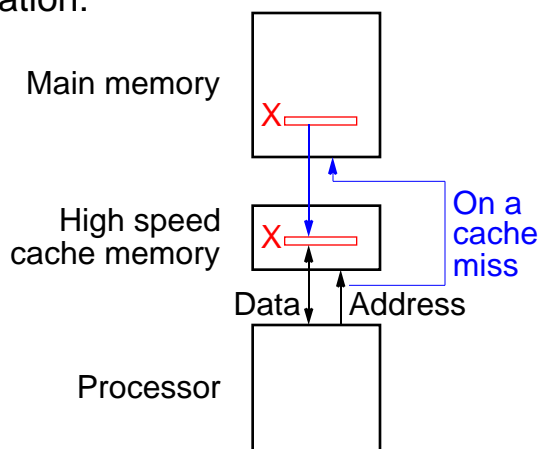
In a practical system, each access time given as an integer number of machine cycles. Can be applied to all equations. Typically hit time will be 1–2 cycles. Cache miss penalty in order of 5–20 cycles.

Cache Memory Organizations

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

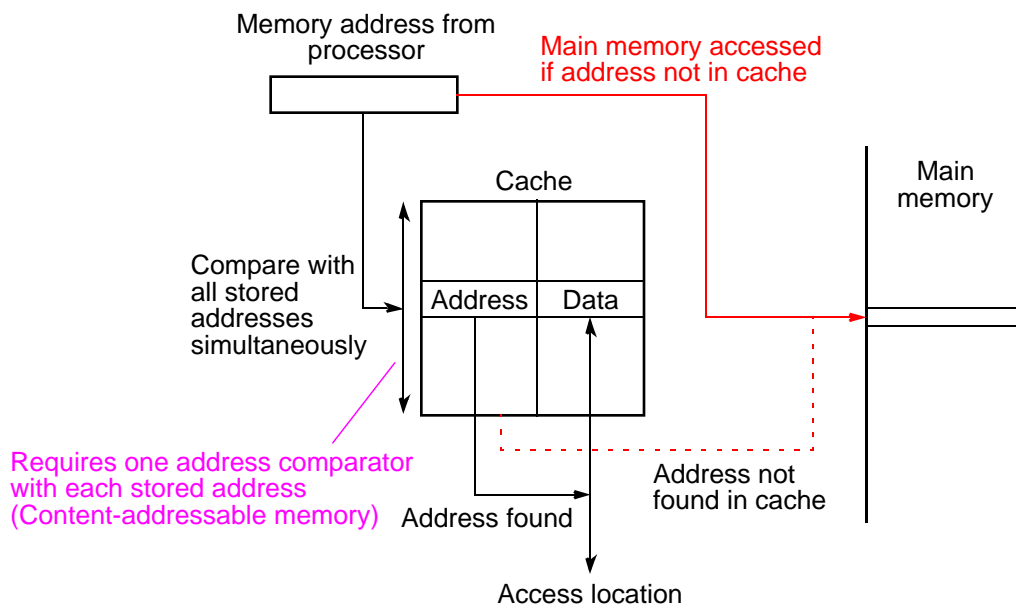
Three ways of selecting cache location:

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

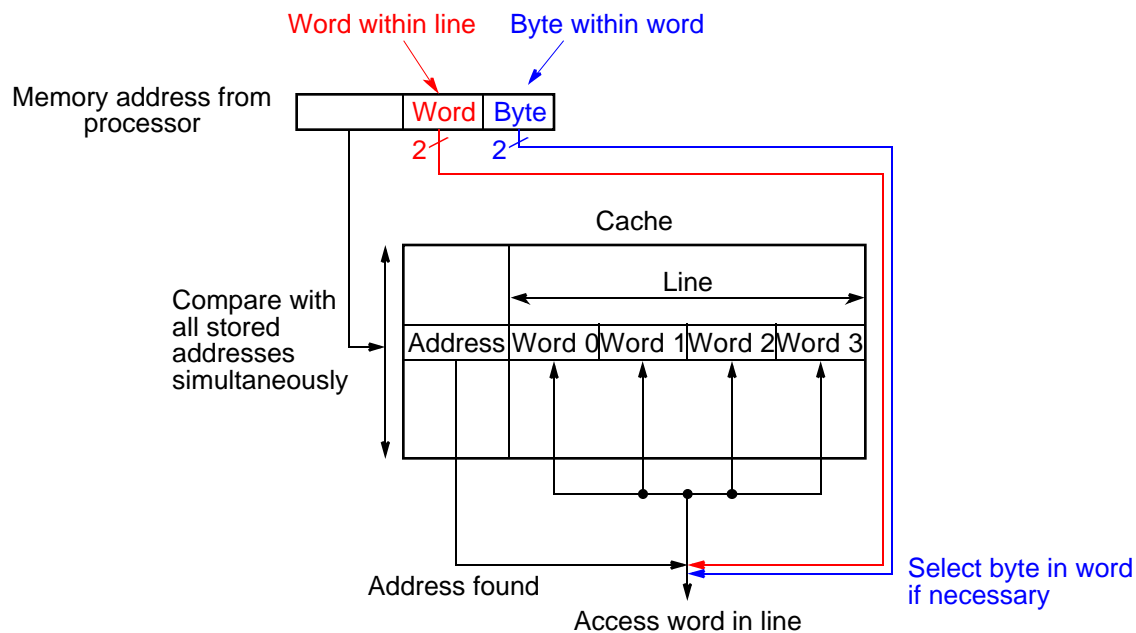


Fully Associative Mapping

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, a word consists of 4 bytes:



Selecting word/byte

In all our figures, address refers to a byte. If processor makes a request for a 32-bit word, appropriate signals generated by processor to select 32-bit word in line. (Similar for other word sizes.)

Pentium Examples

`MOV AL,[100] ;select byte at location 100`

`MOV AX,[100] ;select 16-bit word at 100 and 101`

`MOV EAX,[100];select 32-bit word at 100,101,102,103`

RISCs may not have all these options in selecting size of transfer (or direct/absolute addressing).

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).

Least Recently Used (LRU) Algorithm

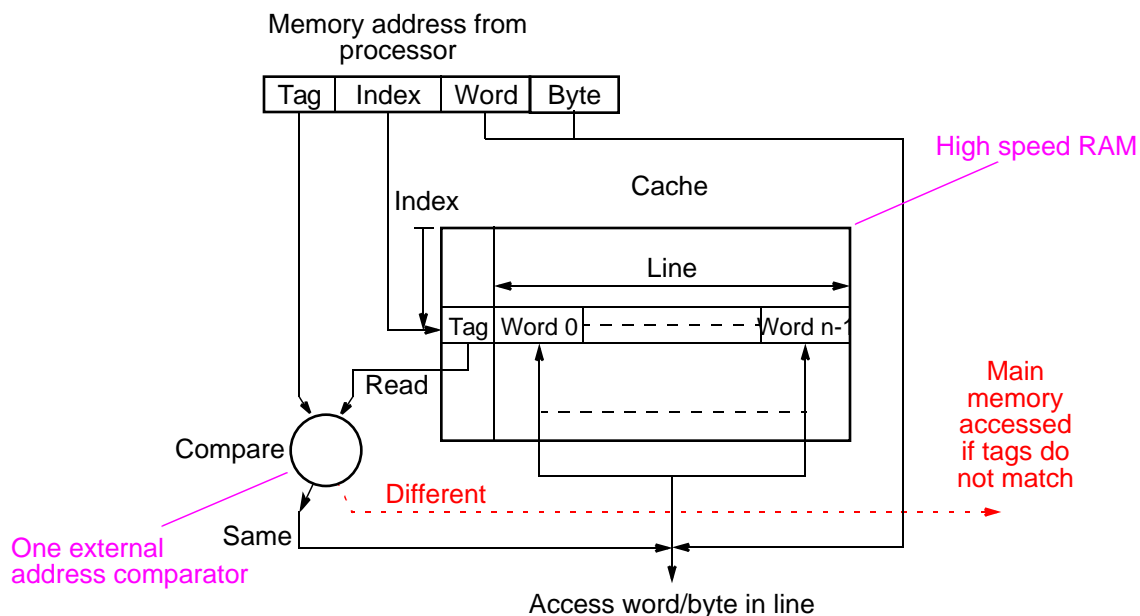
Line which has not been referenced for longest time removed from cache.

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).

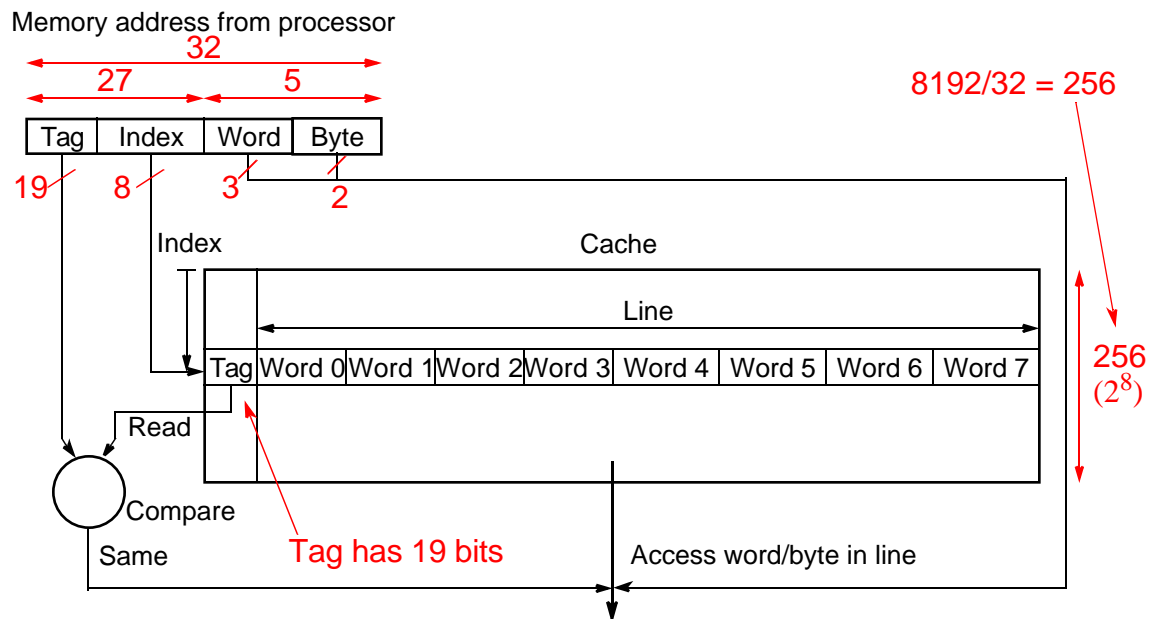
Direct Mapping

Line held in cache at an address given by “index” bits of main memory address. Line selected from “index” bits . Higher significant bits of address (tag) stored in cache:



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 of Direct Mapped Caches

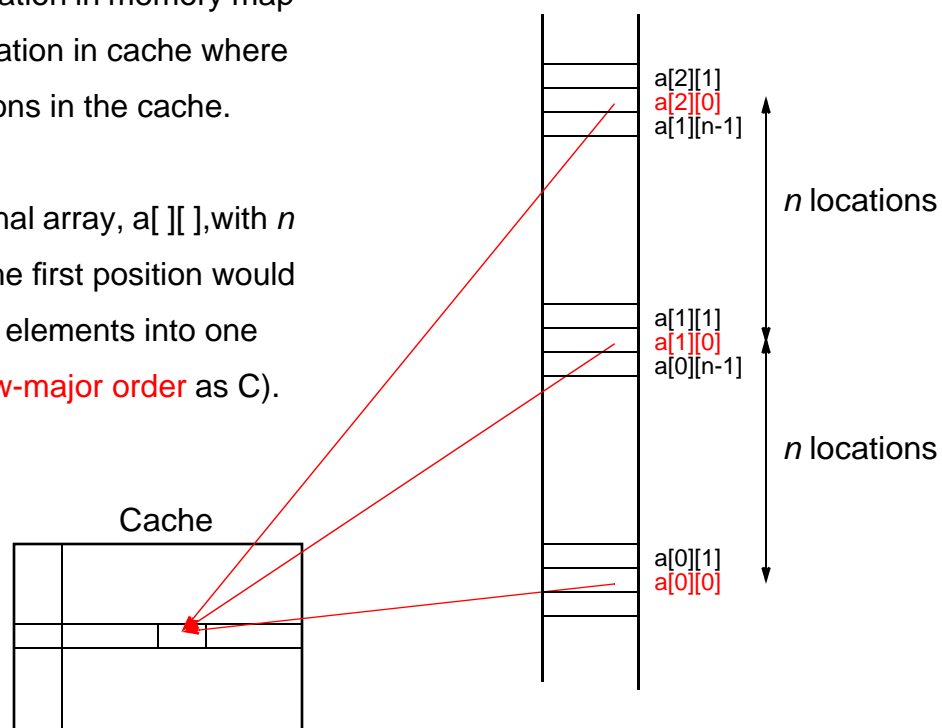
Performance drops significantly if accesses are made to different locations with the same index.

However, as the size of cache increases, the difference in the hit ratios of the direct and associative caches reduces and becomes insignificant.

Elements of an Array Stored in Memory

Every n th location in memory map into same location in cache where there n locations in the cache.

A 2-dimensional array, $a[i][j]$, with n elements in the first position would map all these elements into one location (if **row-major order** as C).



Set-Associative Mapping

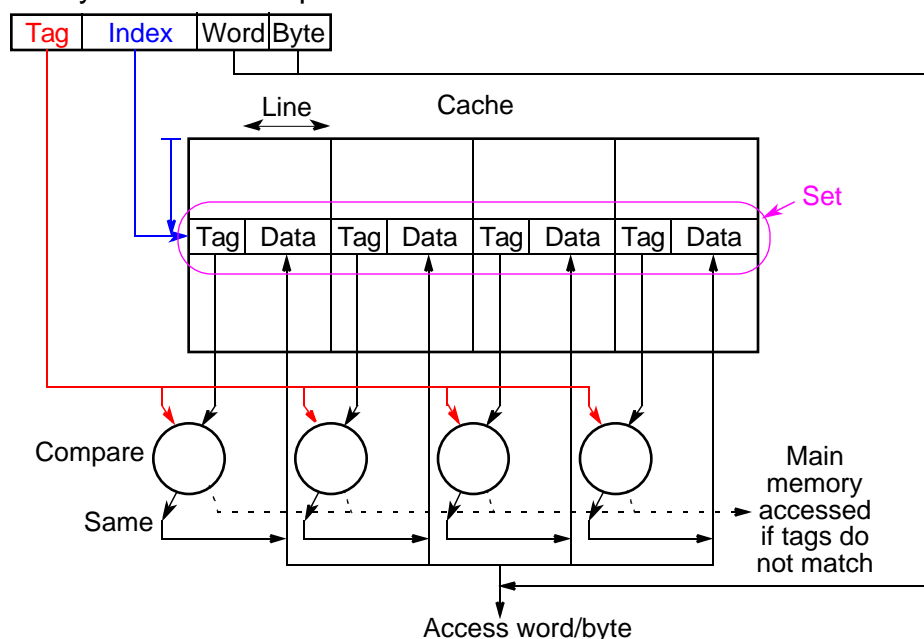
Allows a limited number of lines, with the same index and different tags, in the cache. A compromise between a fully associative cache and a direct mapped cache.

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

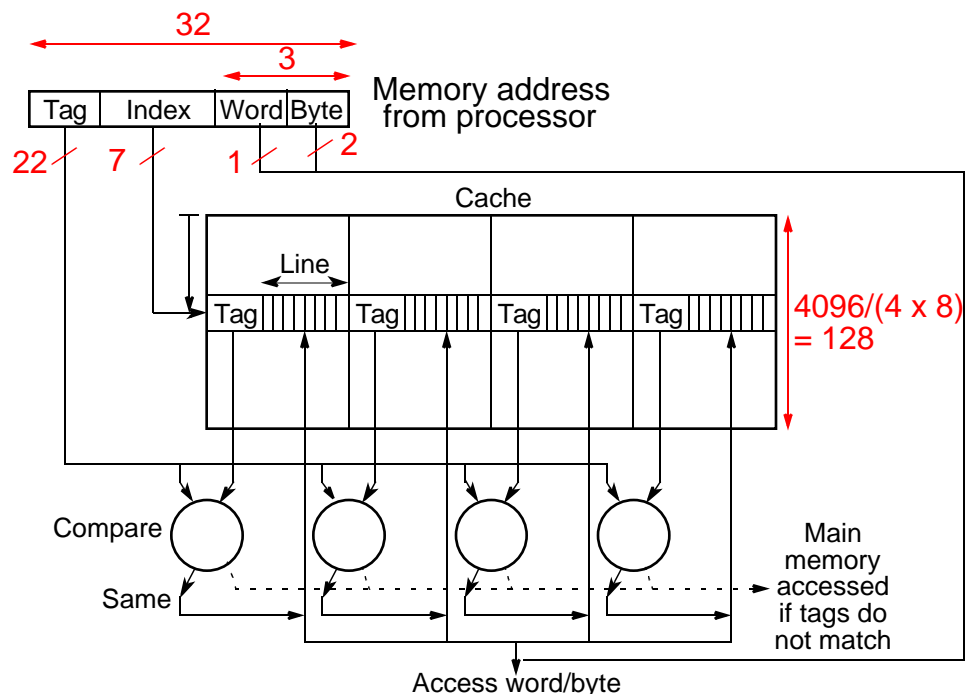
Memory address from processor



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.

Sample 4-way Set-Associative Cache Design

4096-byte 4-way set-associative cache with 8-byte line organized as two 4-byte words. 32-bit memory address.



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, and also before all the cache entries are being used.

If size of line larger than data path between cache and memory

Then, multiple transfers necessary to transfer complete line.

Possible for cache 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 with each part of the line.

Lecture 15

Fetch policy

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

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

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 i th 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 i th 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 Policies

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

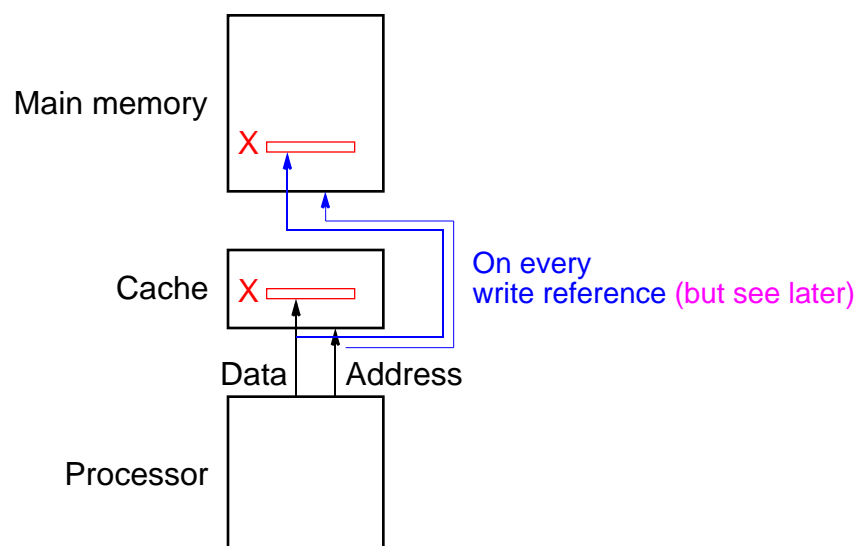
Writing can occur to cache words and then copy held in main memory different. Important to maintain copies same if other devices such as disks access the main memory directly.

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 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 whole line can be transferred 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.

Example

Suppose $t_c = 5$ ns, $t_m = 50$ 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 19 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.

Fetch-on-write (miss)

Describes a policy of 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.

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.

Also called *Non-allocate 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:

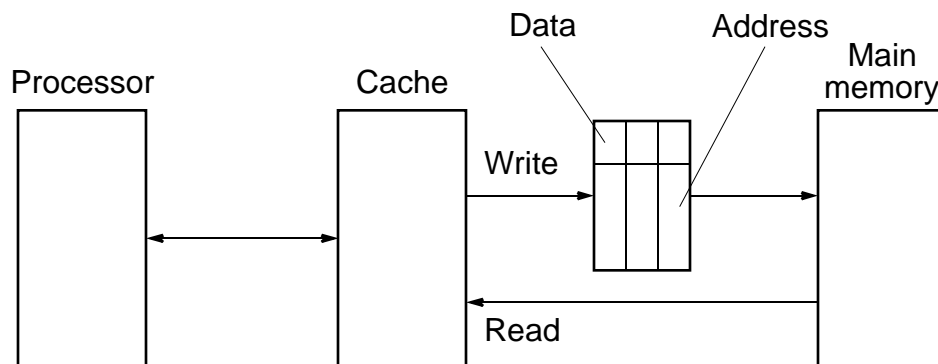
$$t_a = t_c + (1 - w)(1 - h)t_m + w(t_m - t_c)$$

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?**

Cache with write buffer

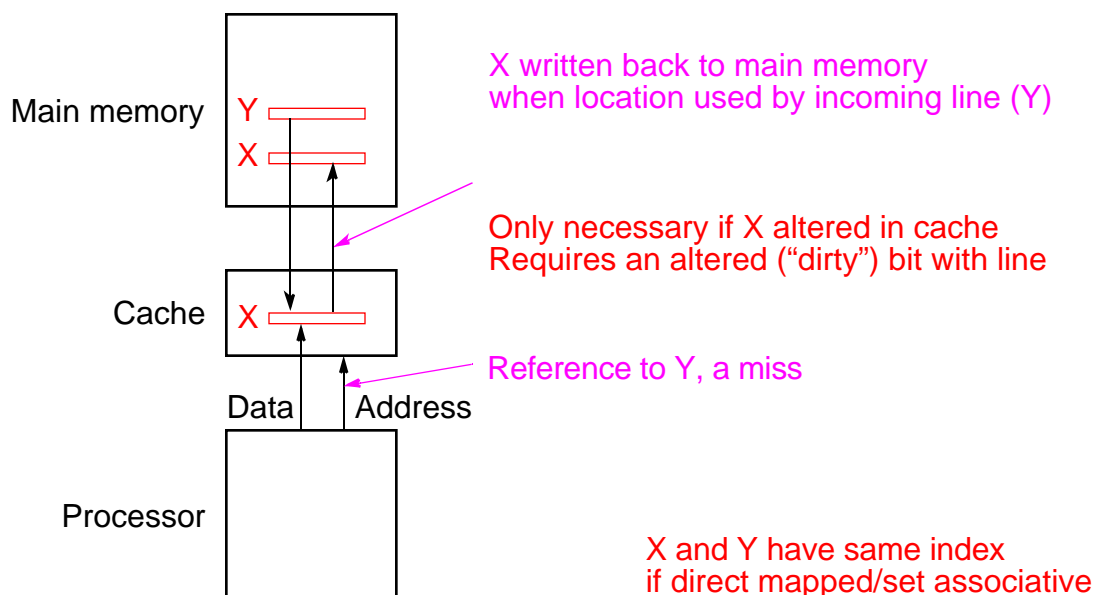
Write-through scheme can be enhanced by incorporating buffers:



Allows the cache to be accessed while multiple previous memory write operations proceed. “Non-blocking” store. E.g. Alpha 21064

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.



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.

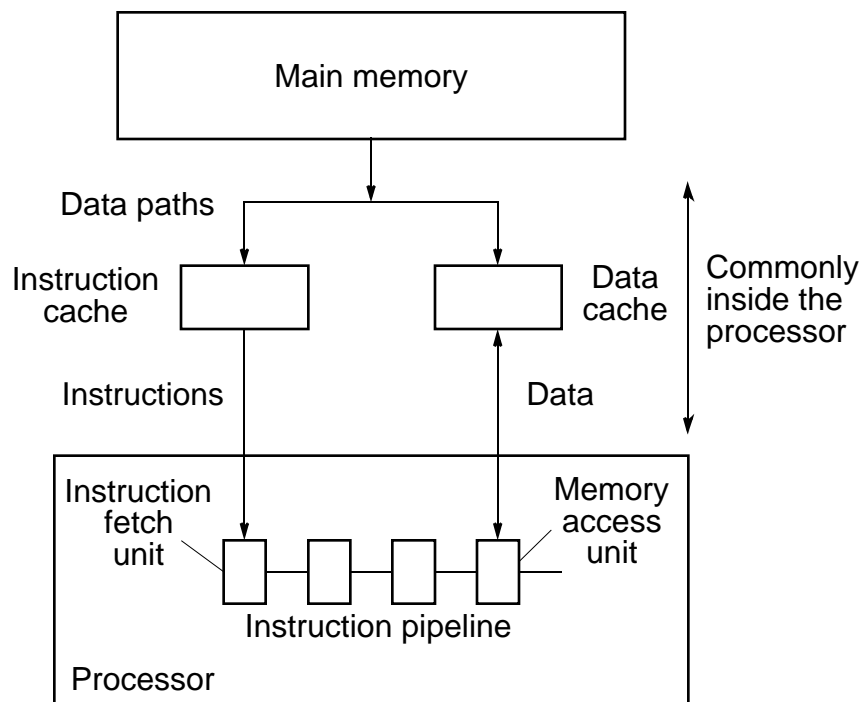
The probability that a line has been altered could be as high as the probability of write references, w , but is likely to be much less, as more than one write reference to the same line is likely and some references to the same byte/word within the line likely.

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):

- Separate paths could be provided from the processor to each cache, allowing simultaneous transfers to both the instruction cache and the data 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.

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



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 fast in hardware.

Ideally, line replaced will not be needed again in the future. Such future events cannot be known and decision has to be made based upon facts that are known at the time.

Classified as usage-based or non-usage-based.

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

Least recently used 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.

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

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 else $B_5 = 0$.

$B_4 = 1$ when line 3 is more recently used than line 1 else $B_4 = 0$.

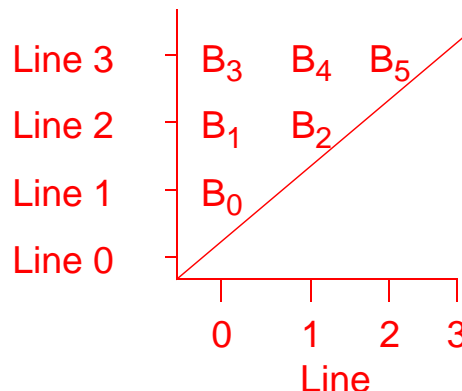
$B_3 = 1$ when line 3 is more recently used than line 0 else $B_3 = 0$.

$B_2 = 1$ when line 2 is more recently used than line 1 else $B_2 = 0$.

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

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

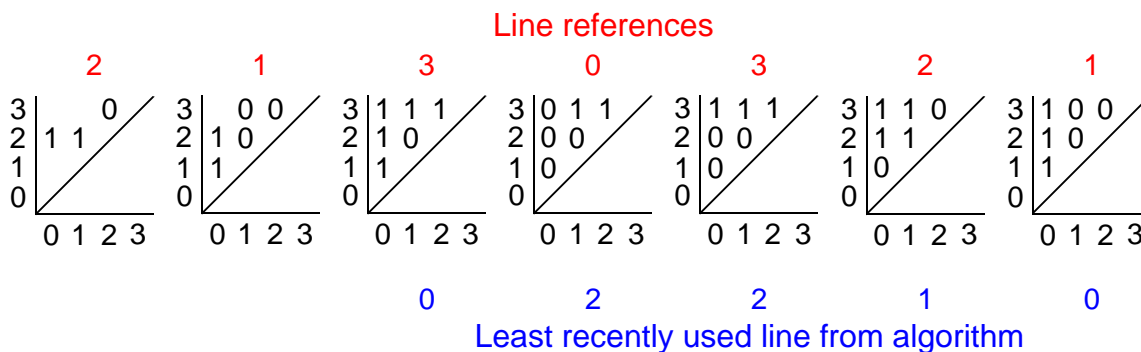
The bits B_5, B_4, B_3, B_2, B_1 and B_0 can be arranged as an upper triangular matrix of a $B \times B$ bits.



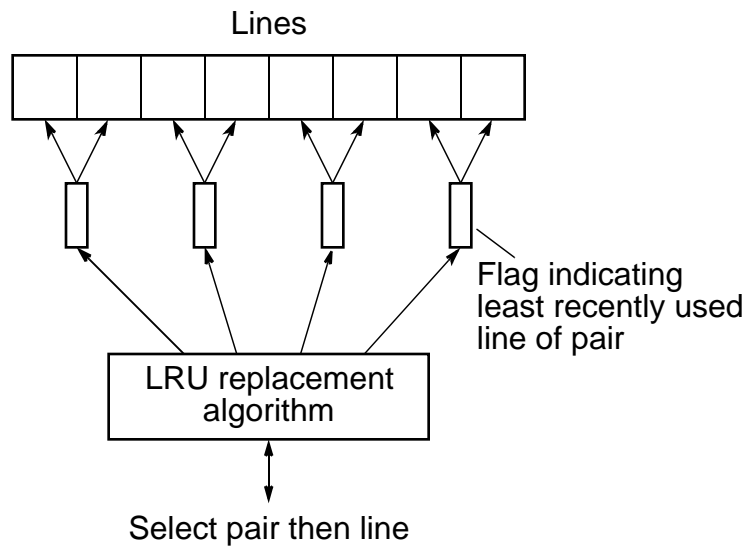
When i th line referenced, all the bits in the i th row of the matrix are set to 1 and then all the bits in the i th column are set to 0. (Prove)

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)

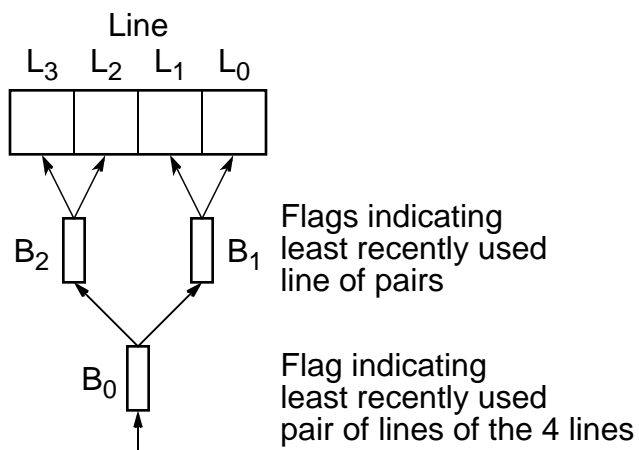
Least recently used replacement algorithm implementation using reference matrix



Two-stage replacement algorithm

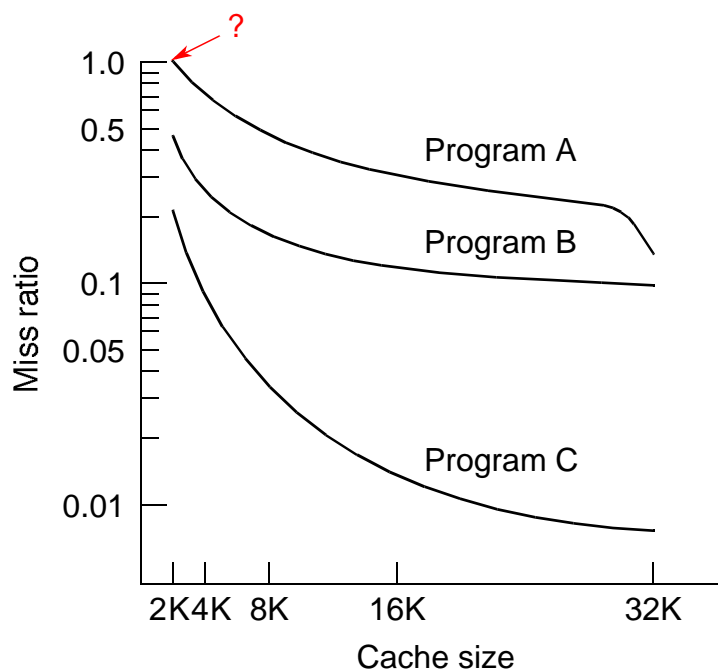


Replacement algorithm using a tree selection

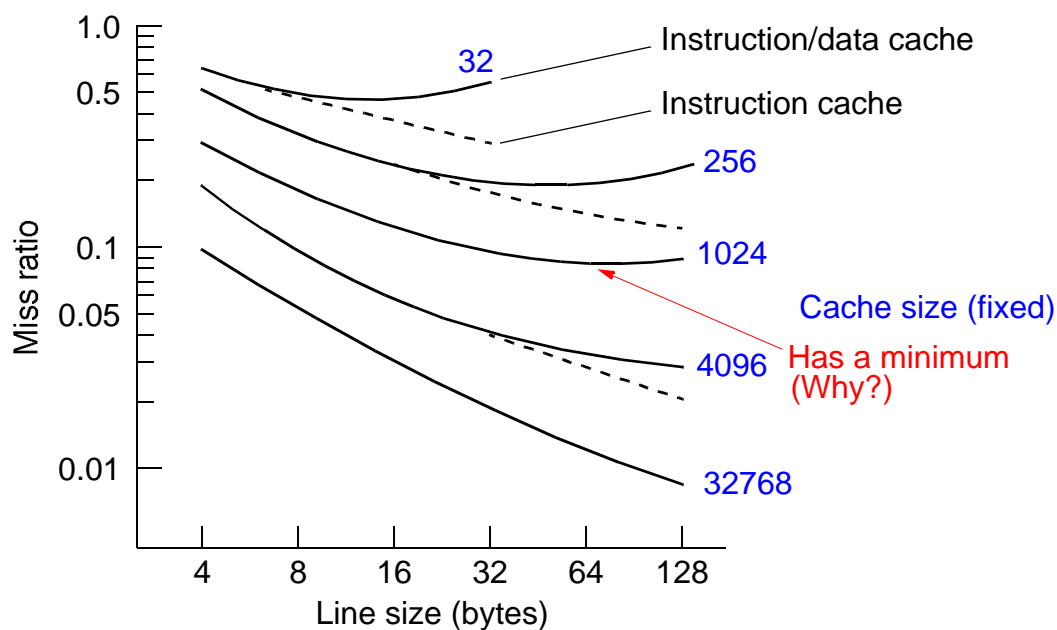


General Cache Performance Characteristics

Miss Ratio against Cache Size

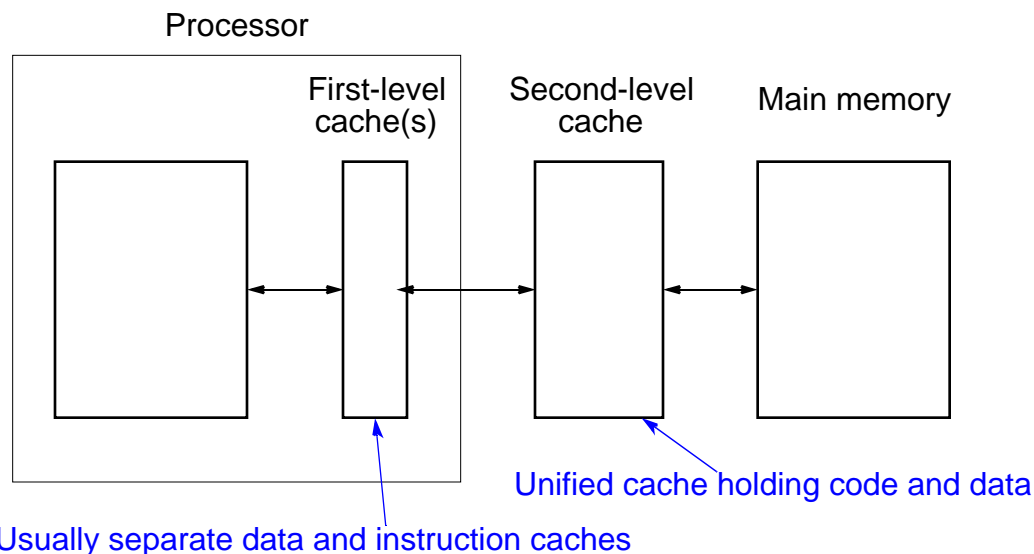


Miss Ratio against Line Size



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:

Second level cache equation

Can extend previous equations to cover 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} = first-level cache access time, t_{c2} = second-level cache access time, t_m main memory access time, h_1 = first-level cache hit ratio, and h_2 = combined first/second-level cache hit ratio, considering two caches as one homogeneous cache system. Most micro-processor families provide for second-level caches.

Lecture 16

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.

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).

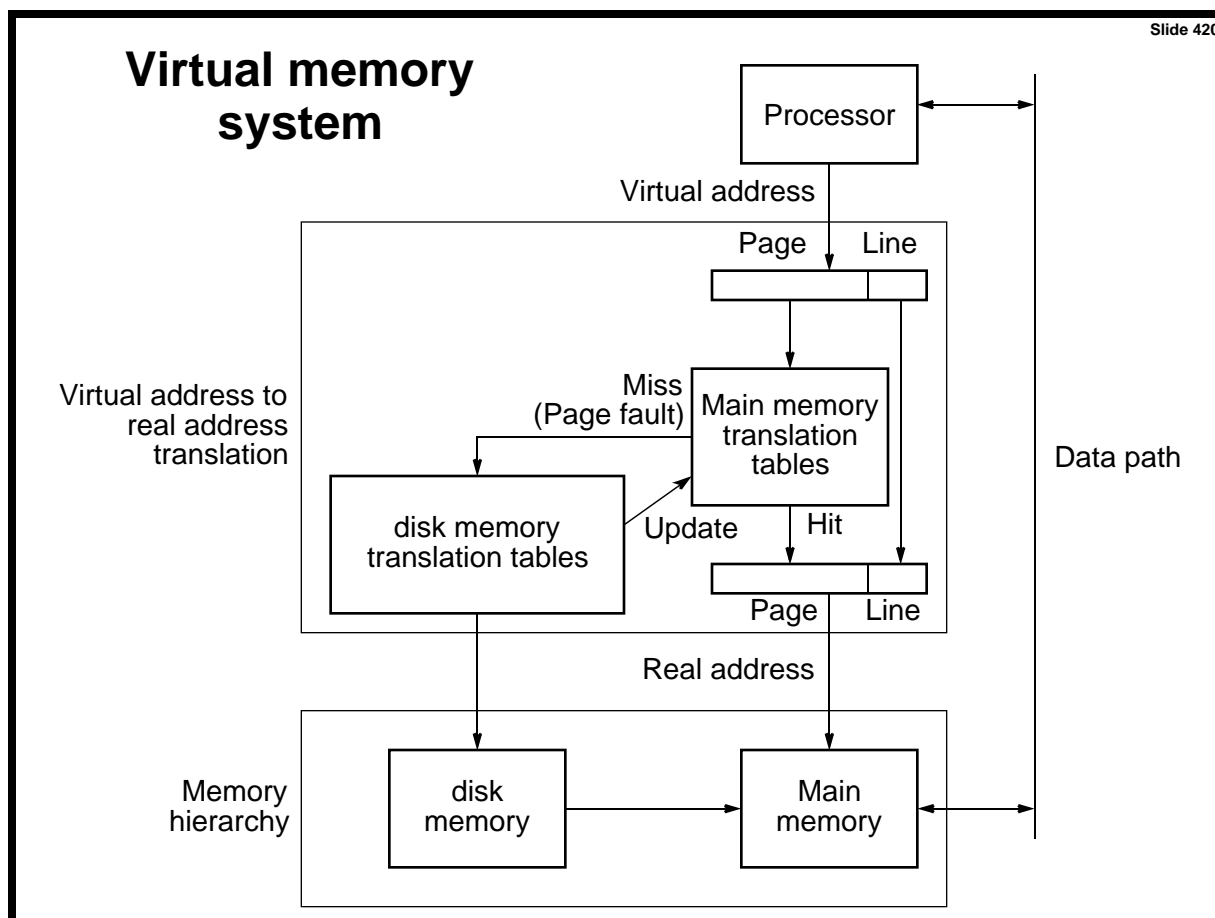
Separate addresses are used for the virtual memory space and the real memory space.

The actual memory addresses are called *real addresses*

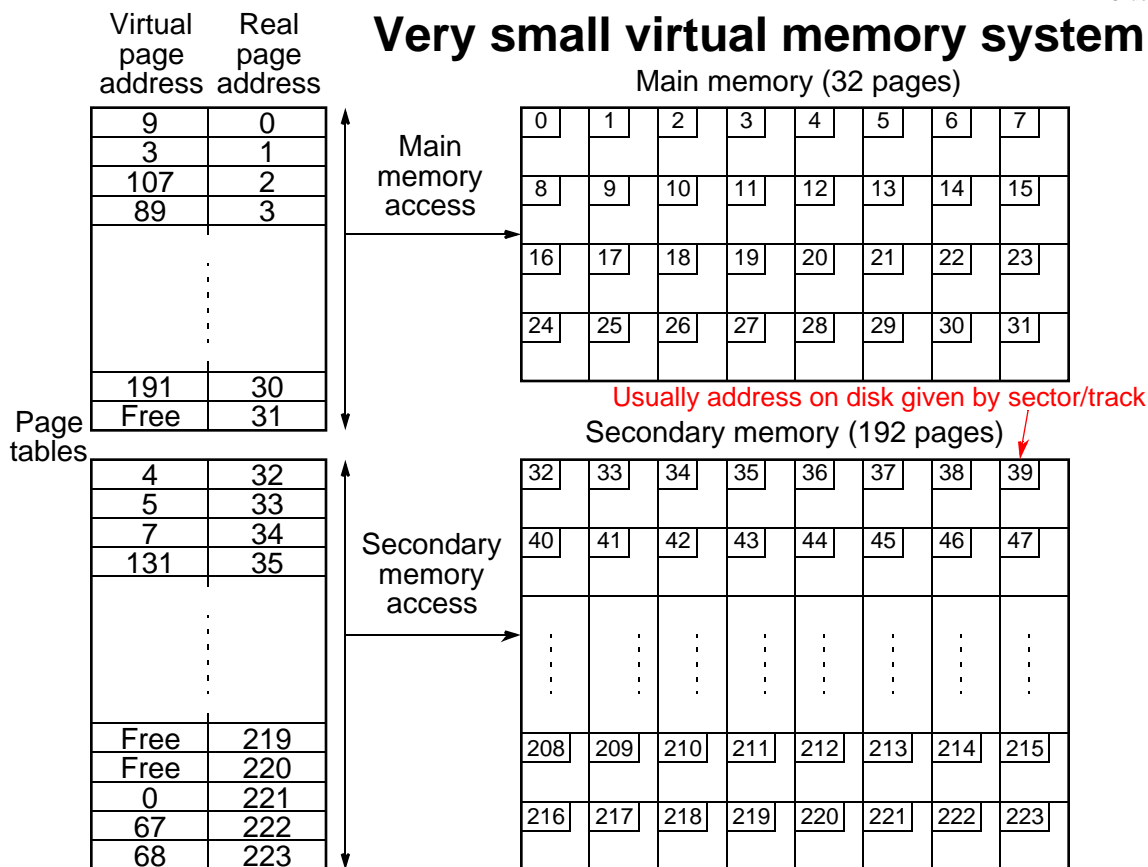
The program generated addresses are called *virtual addresses*.

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

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

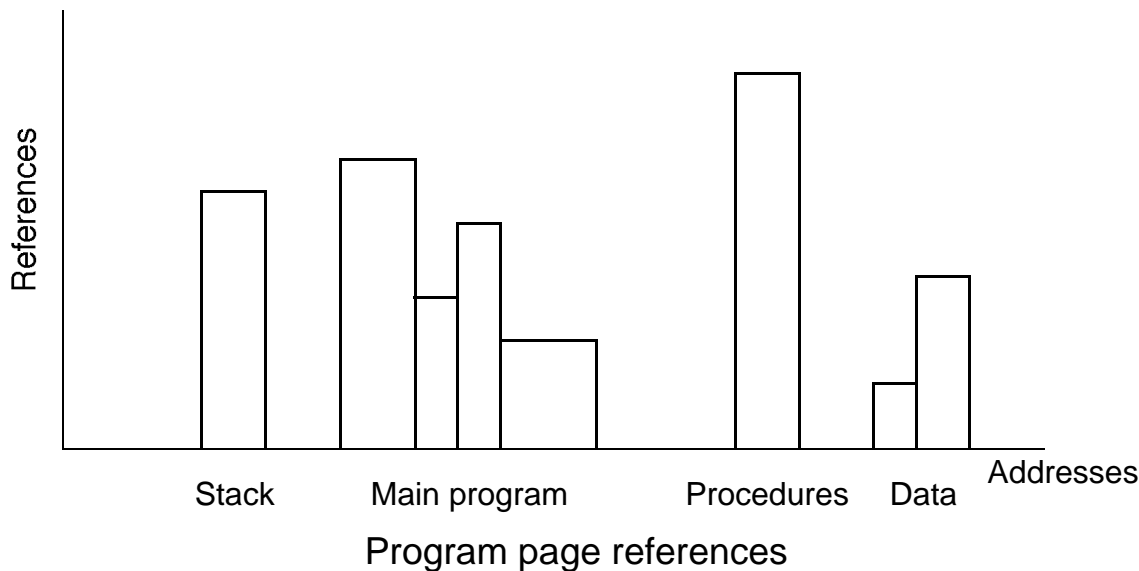


Very small virtual memory system



Principle of Locality

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



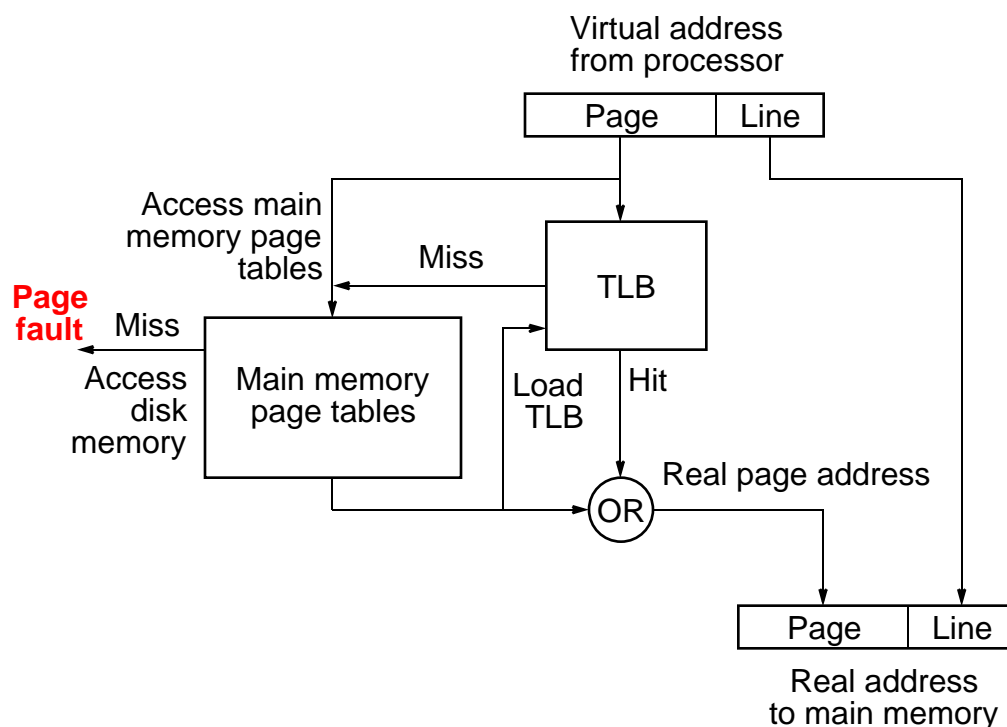
Translation look-aside buffer (TLB)

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 known as a *translation look-aside buffer* (TLB)

Translation look-aside buffer



Data and Instruction TLBs

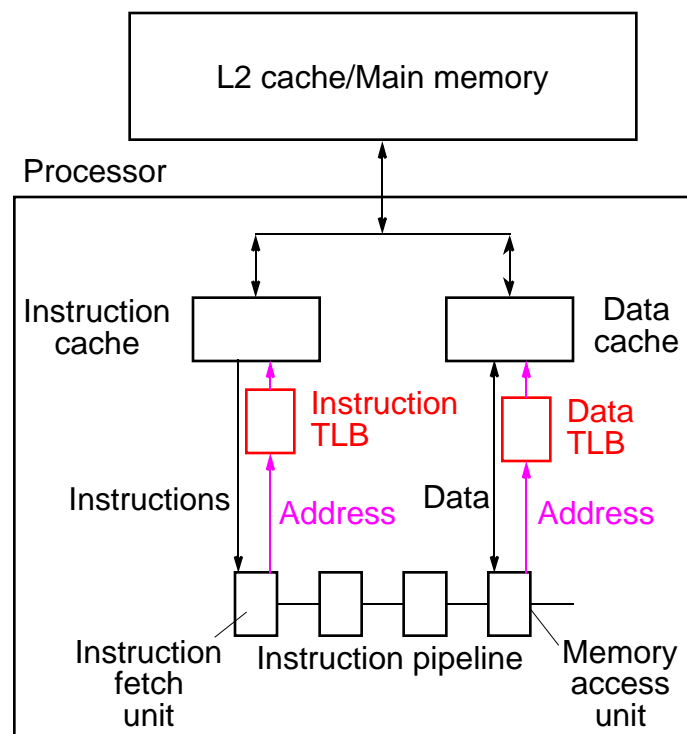
Since most processor have caches and separate data and instruction caches, it is reasonable to use separate TLBs for each type of reference.

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.

Separate TLBs

With caches addressed by real addresses

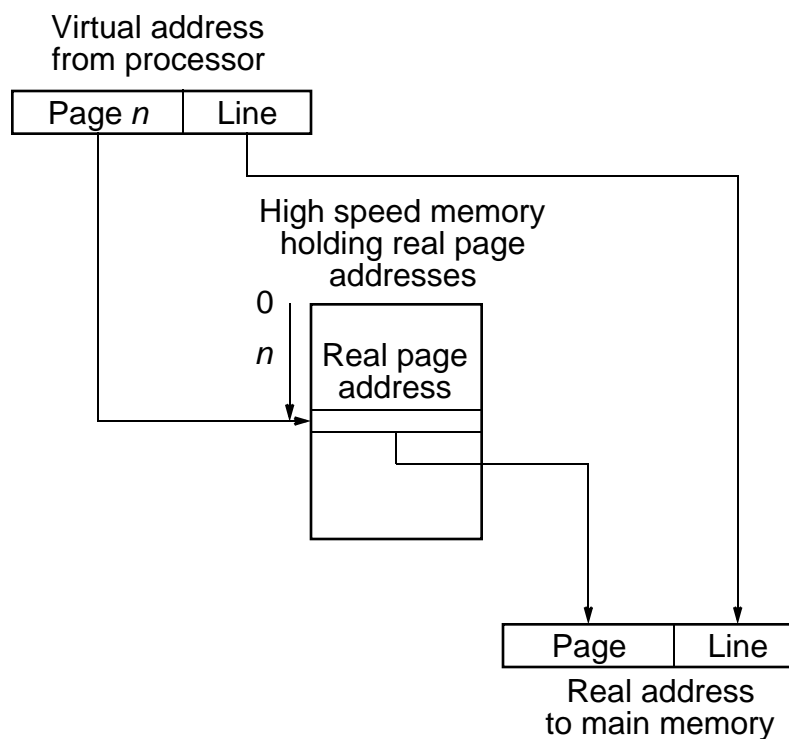


Address translation

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

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

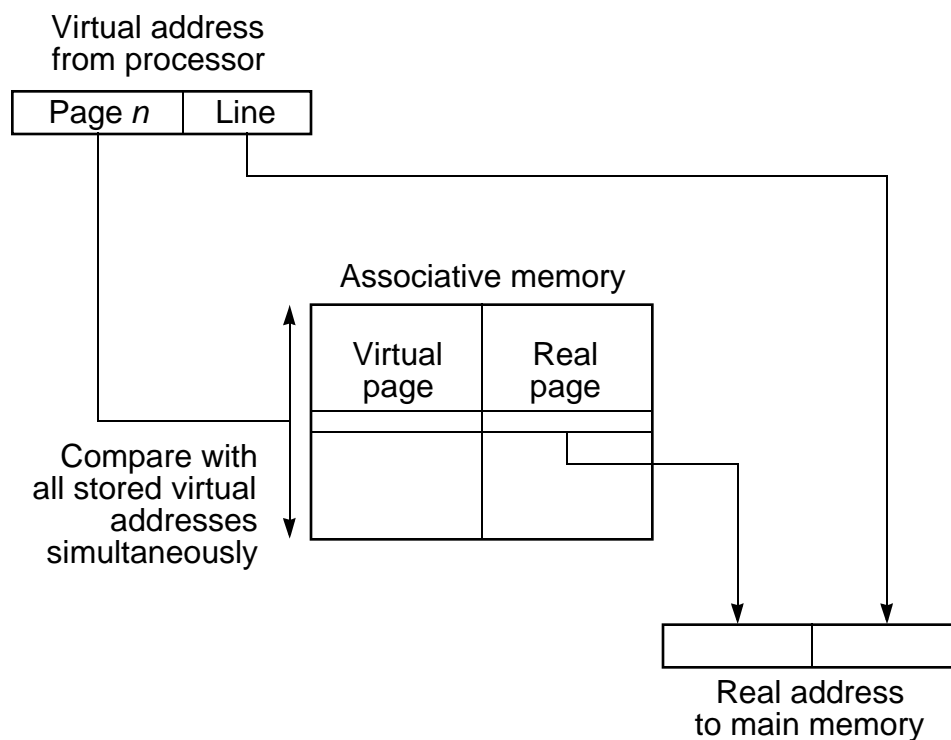
(Pure) direct mapping address translation



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

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

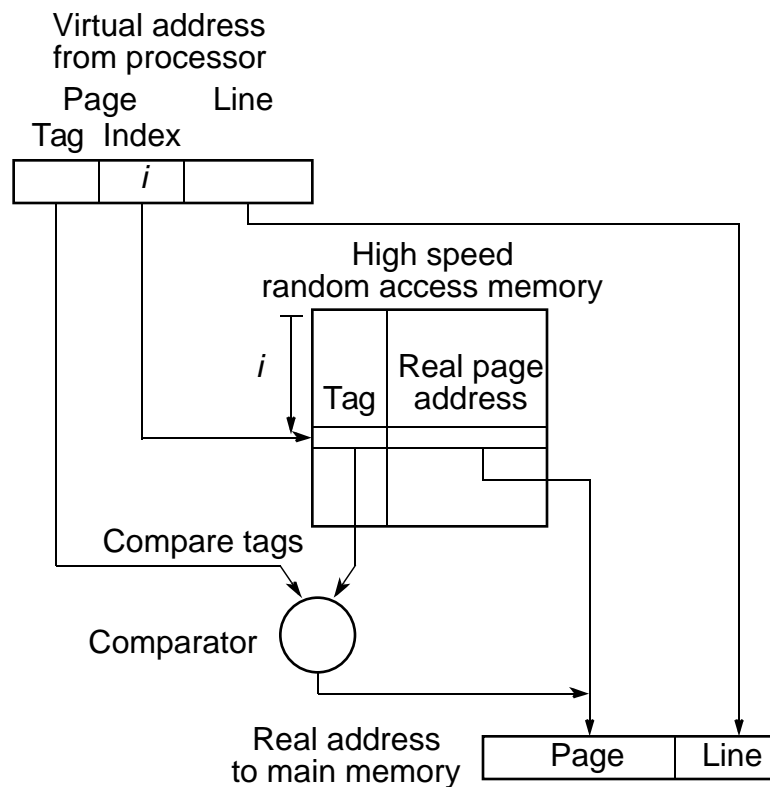
Fully associative mapping address translation



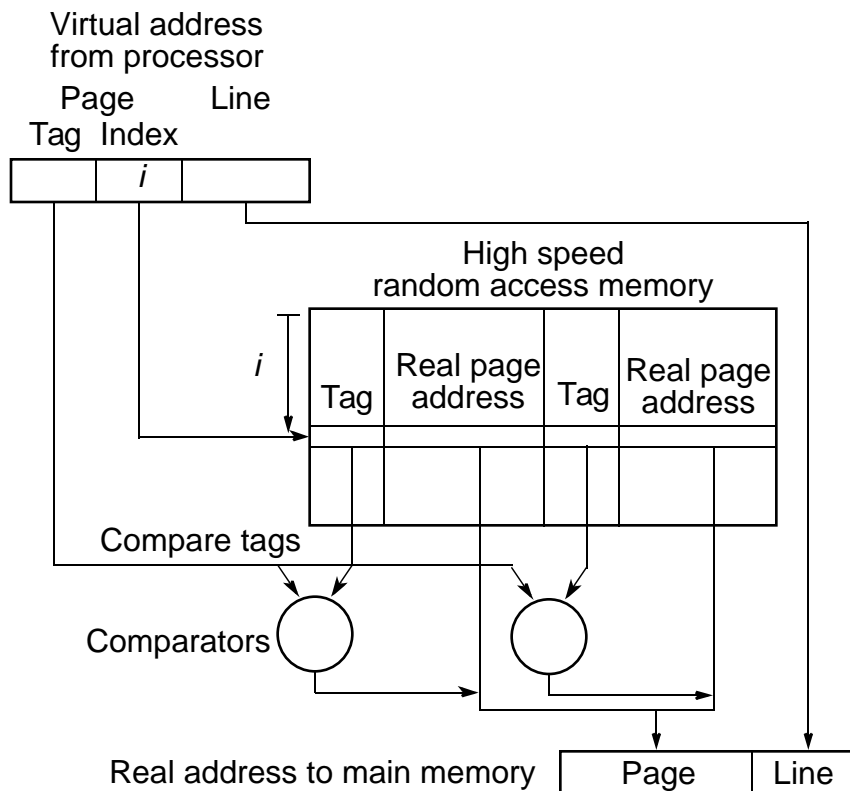
Fully associative method is used in some TLBs.

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.

1-way set-associative mapping address translation



2-way set-associative mapping address translation



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.

Class assignment - find more recent examples.

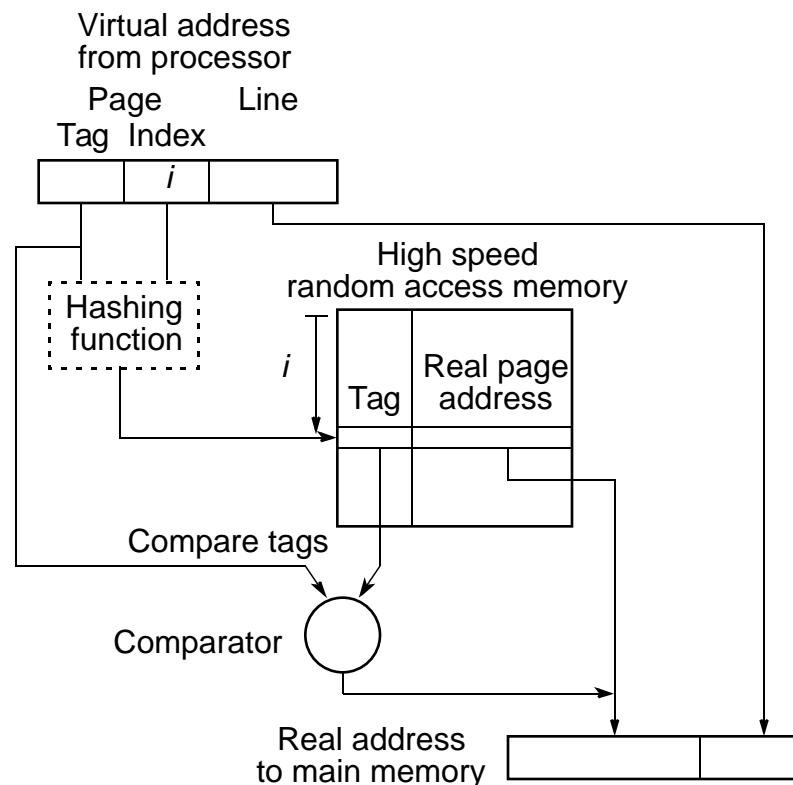
Hashing Functions

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 .

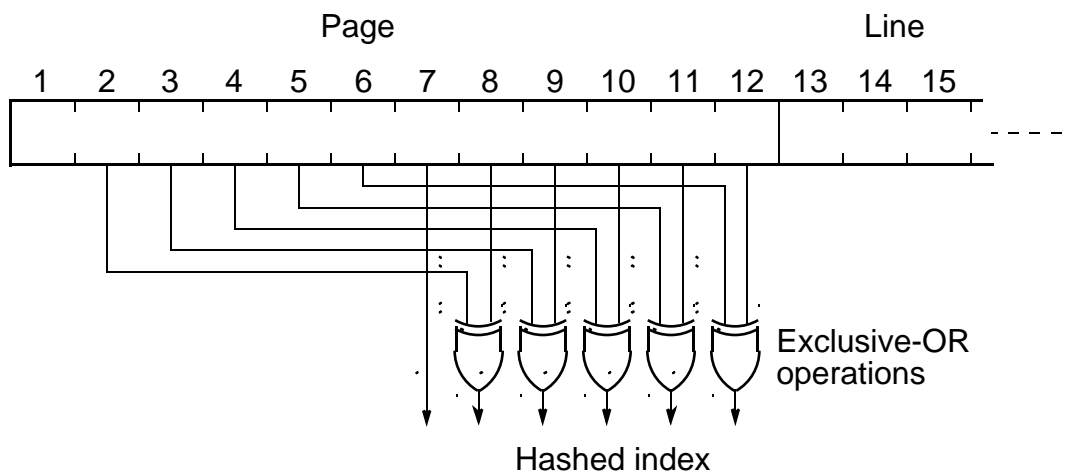
The set size is often only one or two. The chance of virtual addresses having the same lower page bits is quite high.

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

Set-associative mapping address translation (one-way)



IBM 3033 page hashing function



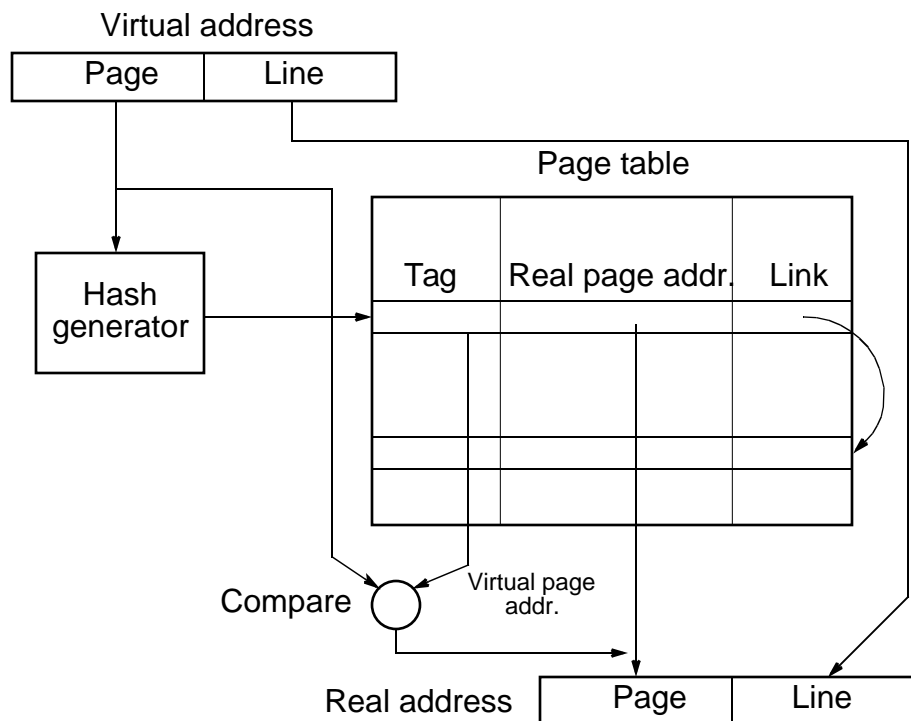
Inverted page tables

A method of reducing the size of the page table (can be applied to the TLB and main page tables).

Rather than store the real address and access table with virtual address, have n entries where there are n pages in main memory, one entry for each real address. Table accessed by hashing virtual address from processor. Need to handle collisions by having pointers to other entries in table.

Method is used in PowerPC.

Inverted page table continued



Inverted page table operation

1. Hash virtual page address
 2. Access inverted page table
 3. Read tag
 4. Compare tag with virtual page address
- If match, read real page address from table
- If no match, follow link to another entry in page table and repeat 2.

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.

Small page size

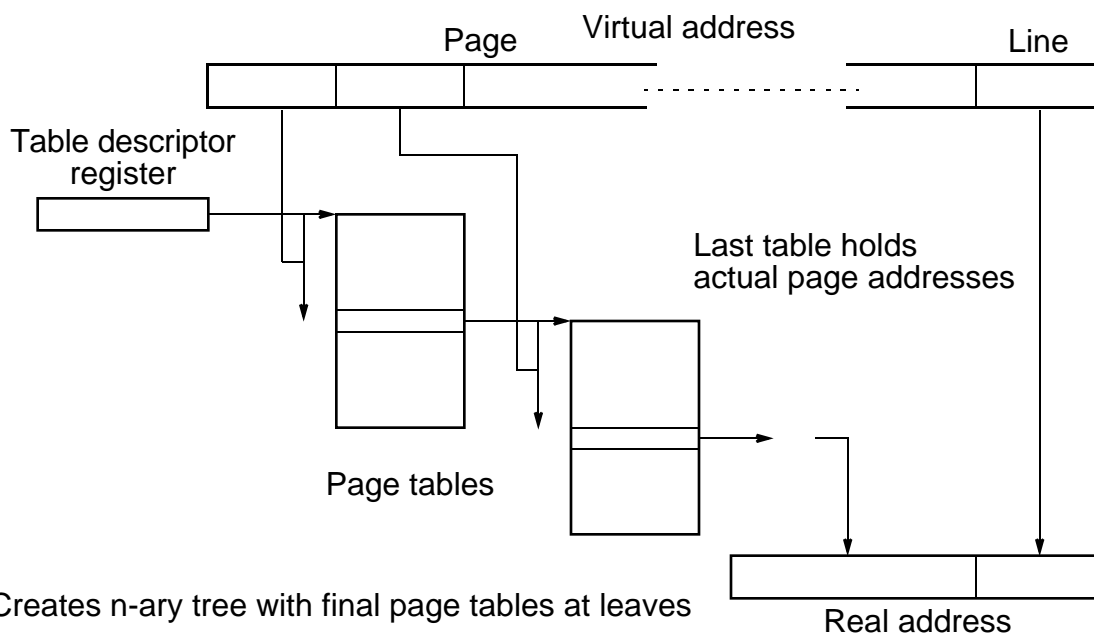
- Time taken in transferring a page between the main memory and the disk memory is short
- Large selection of pages from various programs can reside in main memory.
- Also reduces the storing of superfluous code which is never referenced.
- Necessitates a large page table
- **Table fragmentation** increases. 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.

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.

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.

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.

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 unaltered page back to disk memory if 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.

Least recently used replacement algorithm

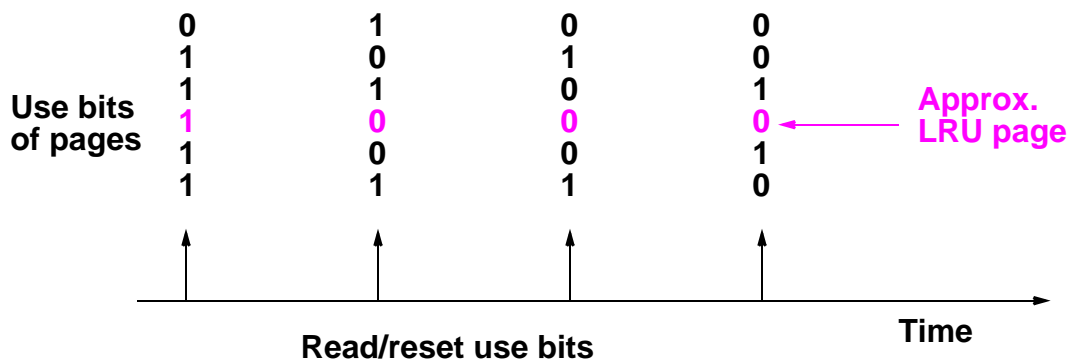
Page not been referenced for longest time transferred out at page fault time. Poses practical problems for a true implementation since there are many pages to consider.

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.

Approximate least recently used replacement algorithm using “sampling”



TLB performance

If the page address not found in 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.

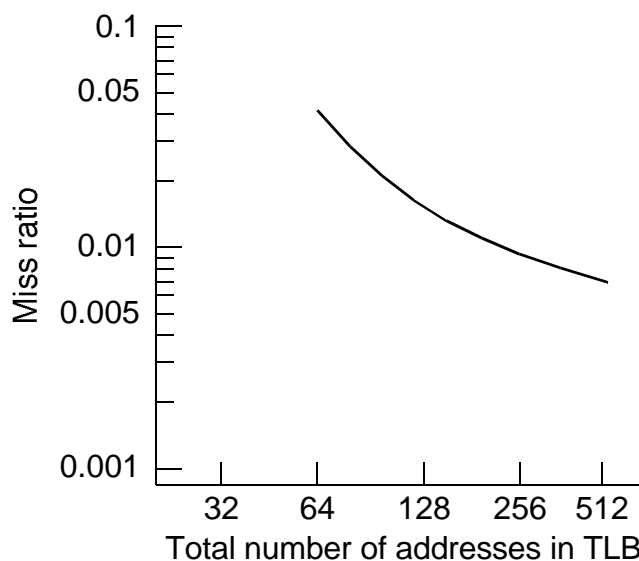
The TLB like a data cache. Basic cache equations also apply to TLB, i.e. the address translation time, t_t , is given by:

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

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.

TLB miss ratio against size



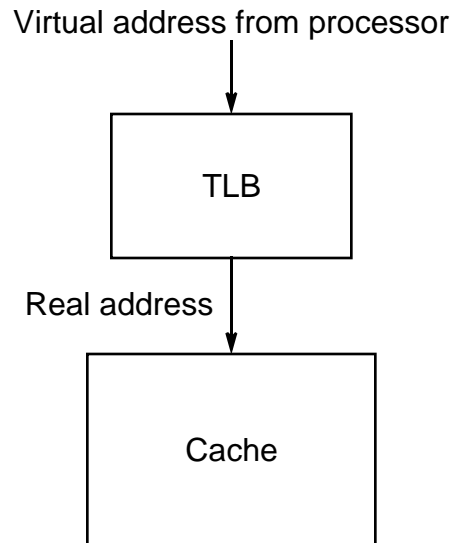
Virtual memory systems with cache memory

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.

Addressing cache with real addresses

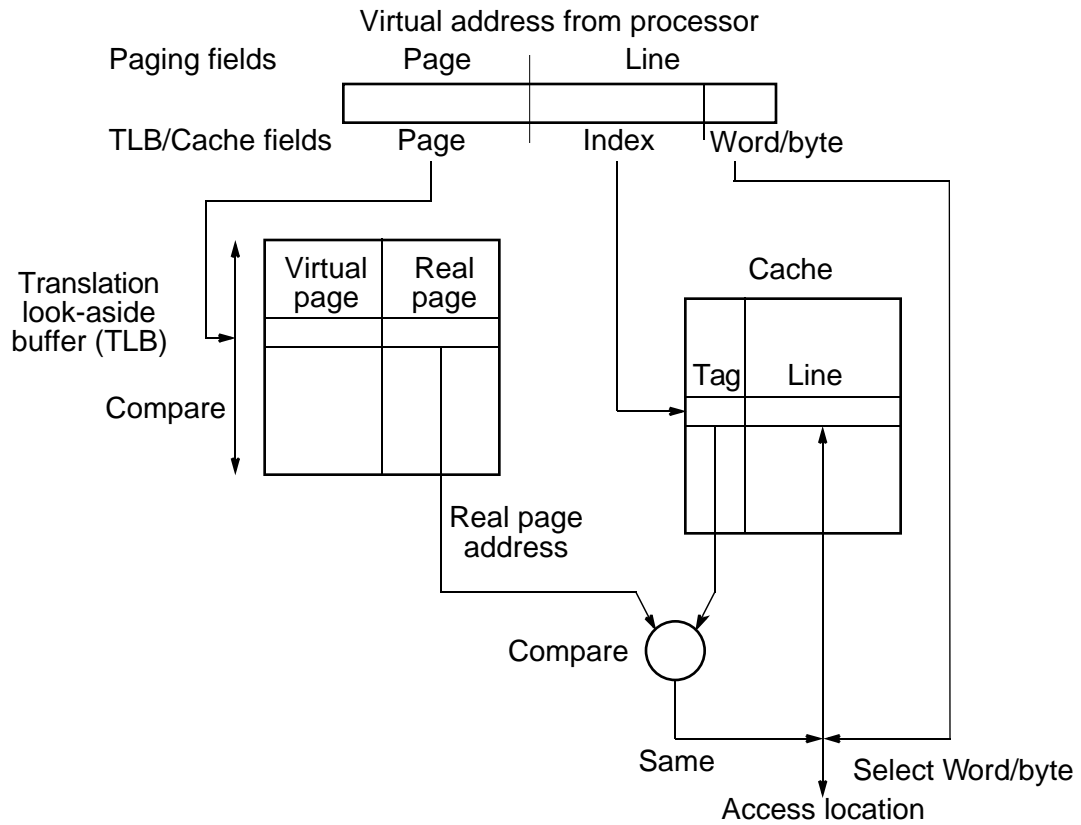


Possible to perform the TLB virtual-real translation at the same time as selecting the line or set in the cache, to gain an improvement in speed. There are constraints on the size of the cache.

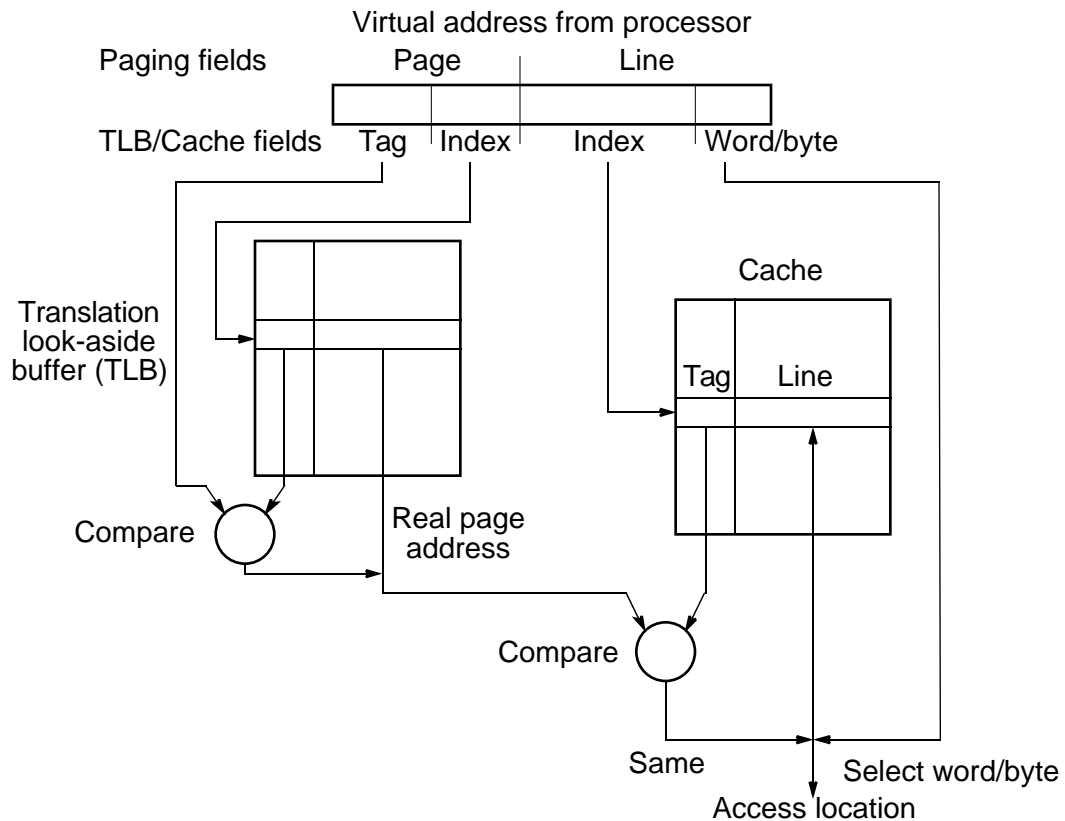
Constraints

Index field has to be contained within the line field of the virtual address - that is, the number of bytes in the cache must be equal or less than the number of bytes in the page - examples next.

Fully associative TLB with a direct mapped cache

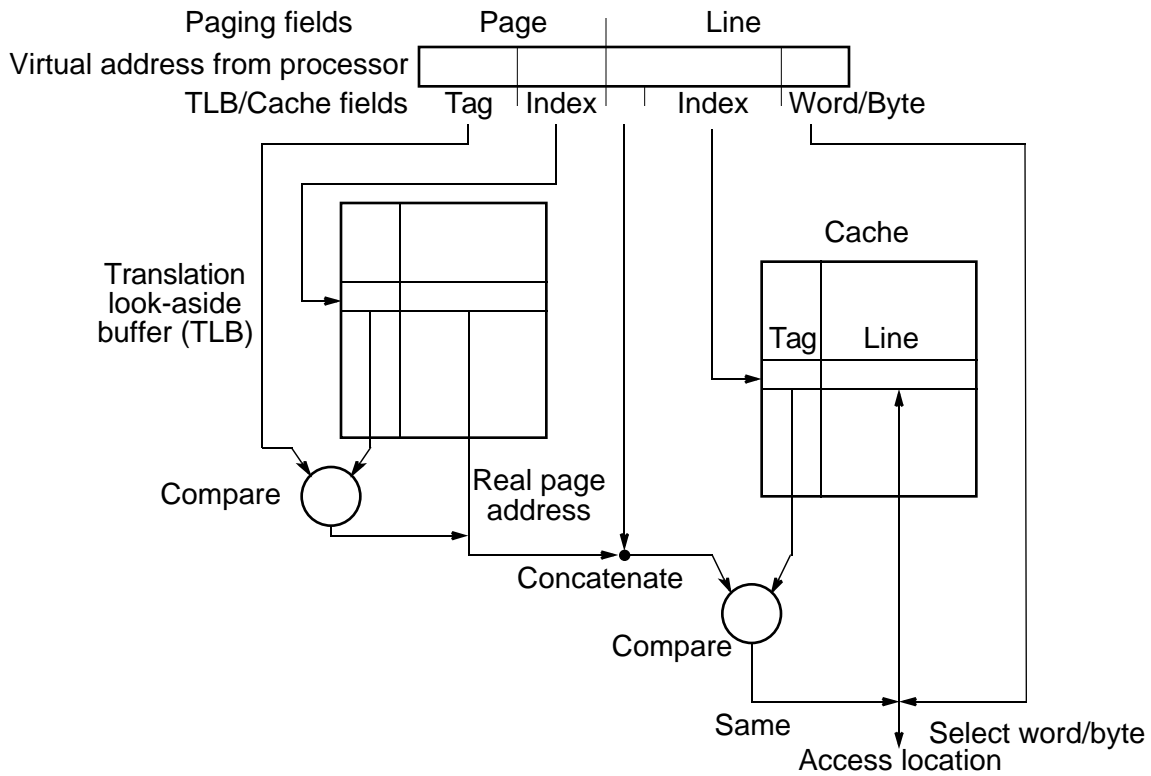


One-way set-associative TLB with a direct mapped cache



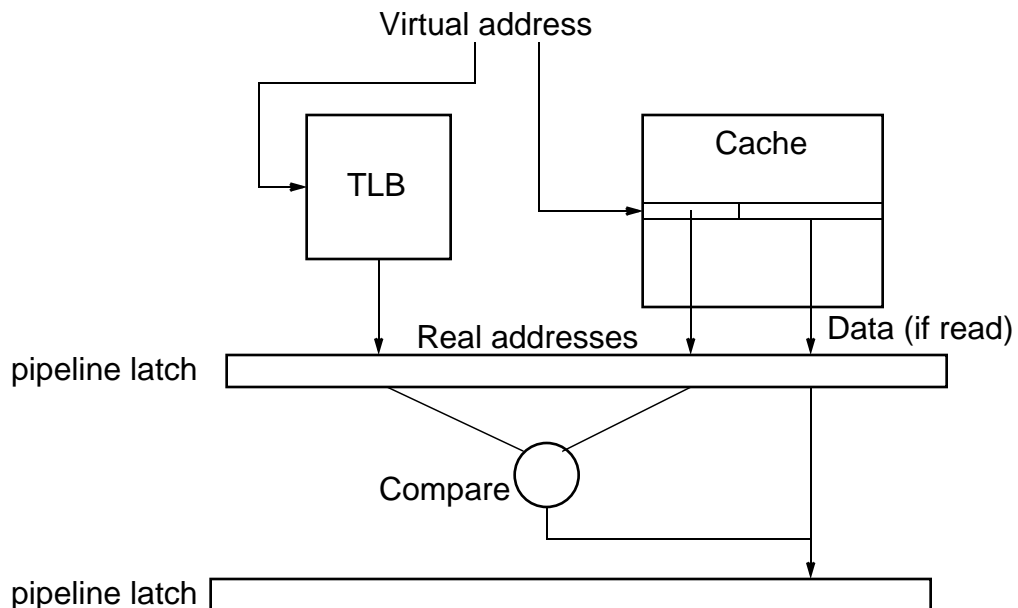
One-way set-associative TLB/direct mapped cache

Larger page size than cache size

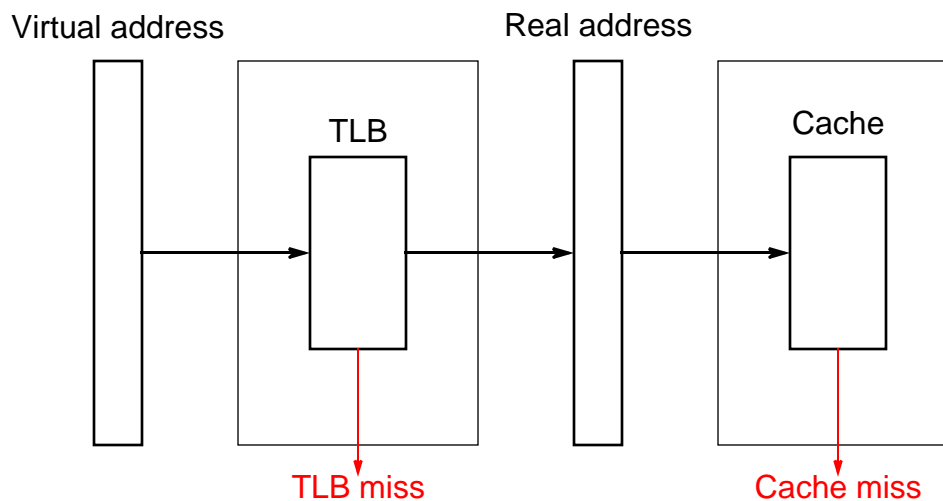


Pipelined Designs

Can pipeline previous designs:



In a pipelined design, could access TLB before the cache and lift previous constraints by having one stage for the TLB and a subsequent stage for the cache:



Notice TLB and cache misses could occur simultaneously relating to different instructions

Real addressed cache access time

At least six combinations of accesses:

	Non-pipelined design
1. Address in TLB, data in cache	$t_{tlb} + t_c$
2. Address in TLB, data in main memory	$t_{tlb} + t_c + t_m$
3. Address in cache, data in cache	$t_{tlb} + t_c + t_c$
4. Address in cache, data in main memory	$t_{tlb} + t_c + t_c + t_m$
5. Address in main memory, data in cache	$t_{tlb} + t_c + t_m + t_c$
6. Address in main memory, data in main memory	$t_{tlb} + t_c + t_m + t_c + t_m$

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* – the same thing as another but in 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.
- Can also occur in multiprocessor systems when processors share memory using different virtual addresses.

Also possible for same virtual address generated in different processes to map into different real addresses.

Synonyms

	Virtual address	Real address
TLB		
	VA1	RA
	VA2	RA

Same

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.

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.

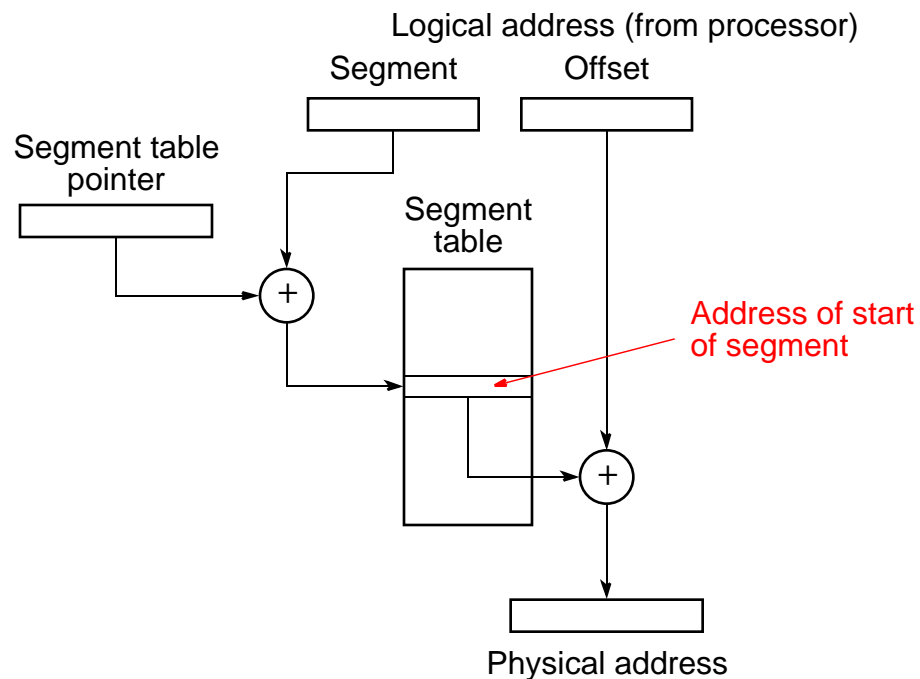
Although the way this is done looks similar to virtual memory/paging, not the same purpose as paging which has a hardware motive, to manage the memory hierarchy in an automatic way.

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 an *offset* within the segment.

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

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, adding one to the offset should create an error condition.

Segment table

Incorporates additional information, including:

1. Segment length.
2. Memory protection bits.
3. Bit(s) for the replacement algorithm.

Segment table Segment start address Segment length Protection bits Use bit

Segment table	Segment start address	Segment length	Protection bits	Use bit

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 logical address is greater than the stored length (limit) field, an error signal is generated (dreaded “segmentation error”).

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).

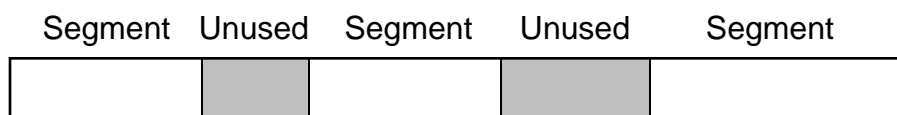
Replacement algorithm

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 bit usually sufficient to implement a replacement algorithm or approximations to a replacement algorithm.

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”:



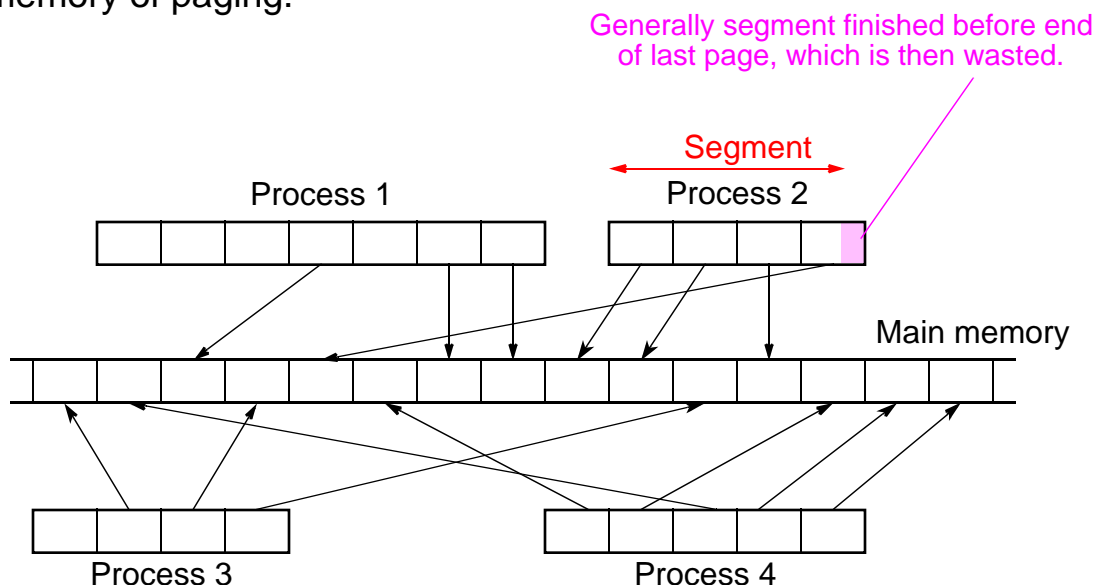
External fragmentation. - leaving small spaces which cannot be used subsequently.

Placement algorithms

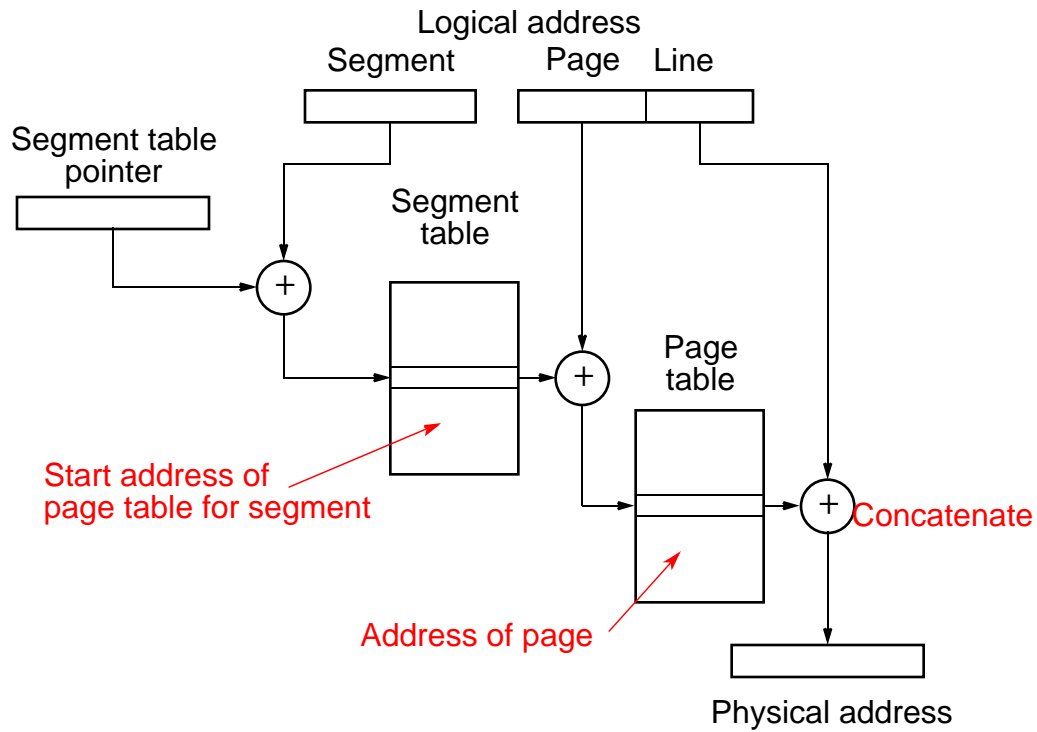
- first fit
- best fit
- worst fit.

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



Page table holds addresses of pages of specific segment given in segment table.

Two-level paging with segmentation

