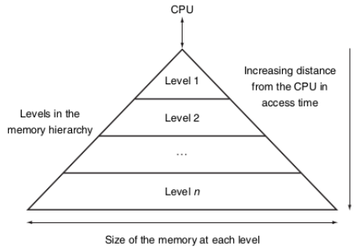
1 Introduction to Memory

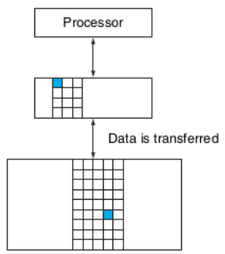


The *principle of locality* states that programs access a relatively small portion of their address space at any instant of time.

- The principle of **temporal locality** states that if a data location is reference then it will tend to be referenced again soon.
- The principle of **spatial locality** states that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

Speed	Size	Cost	Technology
Fast	Small	High	SRAM
	Medium		DRAM
Slow	Big	Low	Magnetic Disk

We take advantage of the principle of locality by implementing the memory of a computer as a **memory hierarchy**, which consists of multiple levels of memory with different *speeds* and *sizes*. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor and spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.



block The minimum unit of information that can be either present or missing in a cache.

hit time The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

miss penalty The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

2 The Basics of Caches

2.1 Direct-mapped Cache

Each memory location is mapped directly to exactly one location in the cache:

(Block address) modulo (Number of blocks in the cache)

Each address is divided into a *tag field*, which is used to compare with the value of the tag field of the cache, and a *cache index*, which is used to select the block. In addition, there is a valid bit indicating whether the cached data is valid.

2.2 Fully Associative Cache

Recall that in a direct-mapped scheme a block can go to exactly one place. The **fully associative scheme** is the other extreme, where a block can be placed in *any* location in the cache. To find a given block in a fully associative cache, all the entries in the cache must be searched. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only for caches with small numbers of blocks.

2.3 Set Associative Cache

The middle range of designs between direct mapped and fully associative is called **set associative**. In a set-associative cache, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called an n-way set-associative cache. An n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique *set*. Thus, a set-associative placement combines direct-mapped placement and fully associative placement: a block is directly mapped into a set, and then all the blocks in the set are searched for a match. In a set-associative cache, the set containing a memory block is given by

(Block number) modulo (Number of *sets* in the cache)

2.4 Choosing Which Block to Replace

The most commonly used scheme is **least recently used (LRU)**, in which the block replaced is the one that has been unused for the longest time.

2.5 Average Memory Access Time

AMAT = Time for a hit + Miss rate × Miss penalty

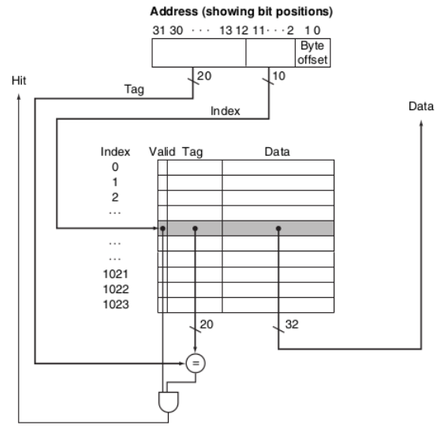


Figure 2.1 A direct-mapped cache holding 1024 words or 4 KiB

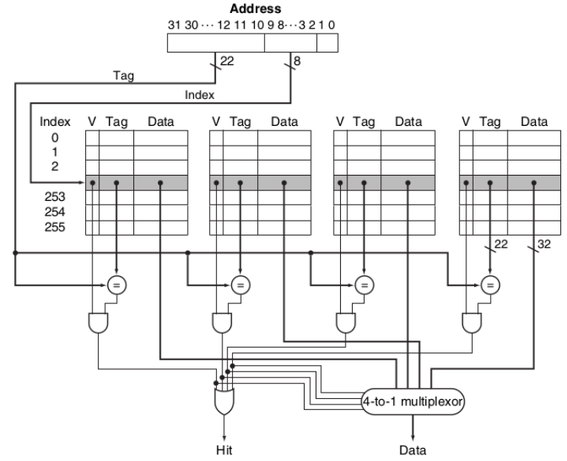


Figure 2.2 A 4-way set-associative cache requires four comparators and a 4-to-1 MUX

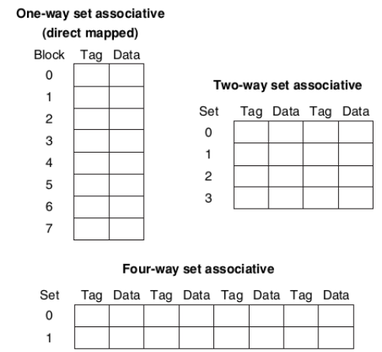


Figure 2.3 An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative

Figure 2.4 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully assoctiave placement

2.6 Handling Cache Misses

When a request for data from the cache that cannot be filled because the data is not present in the cache, the control unit must detect the miss and process it by fetching the requested data from a lower-level cache. This work is done in collaboration with the processor control unit with a separate controller that initiates the memory access and refills the cache. For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory.

1. Send the original PC value (current PC - 4) to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turn the valid bit on.
4. Resume the instruction execution at the first step, which will fetch the instruction, this time finding it in the cache.

2.7 Handling Writes

Suppose on a store instruction, we wrote the data into only the data cache without changing the main memory, then after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*. The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called **write-through**. The alternative is a scheme called **write-back**. When a write occurs, the new value is written only to the block in cache. The modified block is written back to the lower level of the hierarchy when it is replaced. This improves performance but is harder to implement.

2.8 Problem Solving: Calculating Clock Cycle

MIPS Instructions

```
112 addi $1, $0, 100
116 sub $4, $2, $6
120 addi $2, $0, 0
124 lw $5, 0($4)
128 add $2, $5, $2
132 addi $4, $4, 4
136 subi $1, $1, 5
140 bne $1, $0, -5
144 sll $0, $0, 0
```

C Translation

```
$1 = 100; // while guard
$4 = $2 - $6; // array 1st
$2 = 0; // sum counter
while ($1 != 0) {
    $5 = M[$4]; // load elem
    $2 += $5; // update sum
    $4 += 4; // update arr ptr
    $1 -= 5; // update guard
}
```

Assumptions:

1. Forwarding is implemented (so ALU instructions are safe)
2. Flushing is implemented (1 stall for each load-use hazard)
3. Pipeline alreadying running (so no pipeline start-up time)
4. Branching in `ID` stage (1 stall for each branch data hazard)
5. Block size = 4 words (load 4 words into cache each time)
6. Cache begins initially empty (all valid bits are off)
7. Assume once something (data or instruction) is in cache it does not get kicked out of cache

Three Components of the Running Time

1. Pipeline (read **CS251: Processor**)
2. Instruction Cache Misses
3. Data Cache Misses

Pipeline

- Three instructions before the loop (`112-120`).
- Loop runs 20 times; 8 instructions each iteration:
  - Five instructions inside the loop (`124-140`).
  - Load-use hazard requires 1 stall (`124-128`).
  - Branch data hazard requires 1 stall (`136-140`).
  - Branch control hazard requires 1 stall (`144` nop).
- Hence the total running time is $3 + 8 \cdot 20 = 163$.

Instruction Cache Misses

- Examine the first address: is this aligned to the beginning of a block? The block size is 4 and addresses are increments of 4, so is the first address multiple of $4 \times 4 = 16$? Yes.
- The block size is 4 implies each time we make an request, 3 additional (consecutive) words will be brought into cache. We have 9 instructions in total and 112 is a multiple of 16, so we need to fetch $\lceil 9/4 \rceil = 3$ times.
- It takes 104cc to fetch 4 words from RAM. Thus the overall running time for instruction cache misses is $3 \times 104 = 312$.

Data Cache Misses

- Same ideas from instruction cache misses apply here. Moreover, assume data memory accesses are aligned to the first word in the block.
- Recall this program loops over 20 elements of an array and calculate the sum; each array element is brought into cache for use by `lw` at line `124`.
- Since block size is 4 and the data access are aligned, we need to fetch $20/4 = 5$ times in order to get all 20 elements. Therefore, the overall running time for data cache misses is $5 \times 104 = 520$.

Conclusion

- Hence, the total running time when the block size is 4 is $520 + 163 + 312 = 995$.
- Additionally, if the block size is 2, instruction cache misses takes $\lceil 9/2 \rceil \times 102 = 510$ and data cache misses take $10 \times 102 = 1020$. The total number of clock cycle needed is $163 + 510 + 1020 = 1693$.

Other Possible Questions

- No flushing; use rearrangement (remove nops if possible)
- No forwarding (ALU instructions now cause data hazards)
- Branching in `MEM` (3 stalls for branch decision instead of 1)
- Pipeline *not* already running (requires start-up time = 4)
- Cache not initially empty (less fetch from RAM to cache)
- Varying block size (e.g. block size from 4 to 2)
- Varying memory access time (e.g. 102 instead of 104)

Remark on Load vs. Data Cache Miss

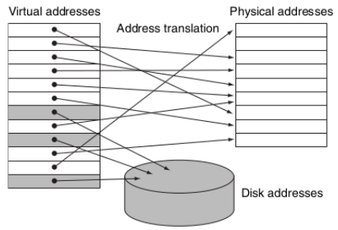
- Data cache is also accessed and modified through `sw`.
- If block size is greater than 1, on a cache miss, we need to first fetch the block from RAM.
- If block size is 1, we do not need to load data from RAM; modifying cache directly is enough.

3 Virtual Memory

3.1 Overview

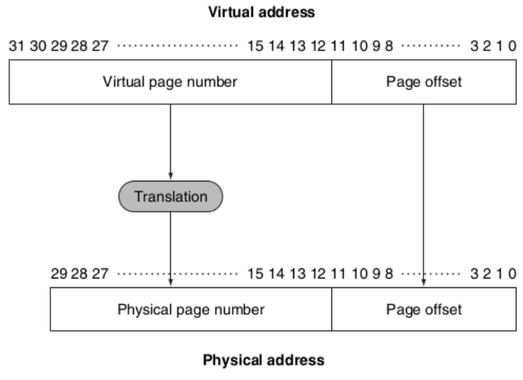
Just as caches provide fast access to recently used portions of a program's code and data, the technique of **virtual memory**, such that the main memory acts as a "cache" for the secondary storage, allows efficient and safe sharing of memory among multiple programs (e.g. for cloud computing) and removes the size limit of main memory. To ensure that a program can only read and write the portions of memory that have been assigned to it, we would like to compile each program into its own *address space* – a separate range of memory locations accessible only to this program.

A virtual memory block is called a **page**; a virtual memory miss is called a **page fault**. With virtual memory, the processor produces a **virtual address** that is translated by a combination of hardware and software to a **physical address**, which in turn can be used to access main memory. This process is called *address mapping* or **address translation**.



3.2 Address Translation

In virtual memory, the address is broken into a **virtual page number** and a **page offset**. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address (length of VPN) need not match the number of pages addressable with the physical address (length of PPN), thus you can have a large number of virtual pages and create an illusion of an essentially unbounded amount of virtual memory.

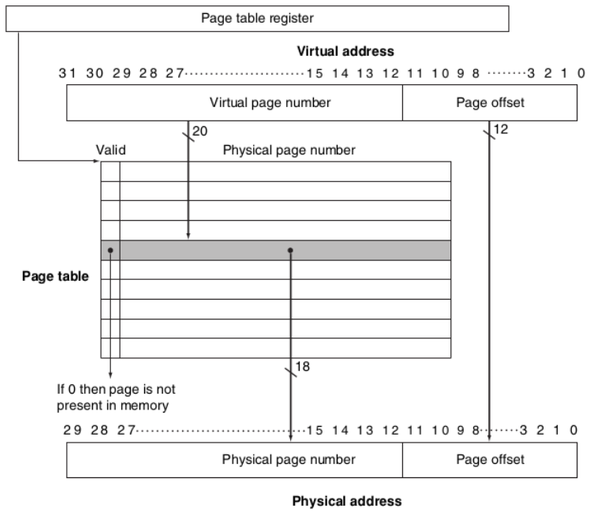


3.3 Page Table

The **page table** contains the virtual to physical address translations in a virtual memory system. The table, which itself is stored in RAM, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

The page table, together with the PC and the registers, specifies the **state** of the program. To allow another program to use the processor, we must save this state. Later, after restoring this state, the program can continue execution. We often refer to this state as a **process**; it is considered *active* when it is in possession of the processor; otherwise, it is considered *inactive*. The OS can make a process active by loading the process's state, including the PC, which will initiate execution at the value of the saved PC.

The process's address space, and hence all the data it can access in memory, is defined by its page table, which resides in RAM. Rather than save the entire page table, the OS simply loads the **page table register** to point to the page table of the process it wants to make active. Each process has its own page table, since different processes use the same virtual addresses. The OS is responsible for allocating the physical memory and updating the table tables, so that the virtual address spaces of different processes do not collide.



- The page table register leads us to this page table.
- The valid bit indicates whether the data of this entry is valid.
- The 12-bit page offset stays the same after translation.
- Each 20-bit VPN is mapped to an 18-bit PPN in the table.
- Given a VA, we go to the page table pointed to by PTR and look up the 20-bit VPN. If the entry exists and the valid bit is on, we concatenate the corresponding PPN with the page offset to produce PA.

3.4 Page Faults

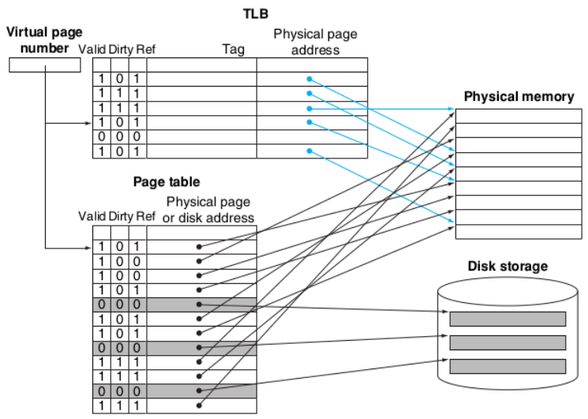
If the valid bit for a virtual page is off, or the entry for that VA does not exist in the page table, a page fault occurs; the OS must be given control. The OS also creates a data structure that tracks which processes and which virtual addreses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the OS must choose a page to replace. Using the LRU scheme, the replace pages are written to **swap space**, the space on the disk reserved for the full virtual memory space of a process.

Implementing a completely accurate LRU scheme is too expensive, thus most OS approximate LRU with a **reference bit**, which is set whenever a page is accessed. The OS periodically clears the reference bit and later records them so it can determines which pages were used during a particular time period. With this usage info, the OS can select a page that is among the least recently referenced to be replaced.

3.5 Translation-Lookaside Buffer

Since the page tables are stored in RAM, each memory access by a program can take at least twice as long: one memory access to obtain the PA and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. Accordingly, most modern processors include a special *cache* that keeps track of recently used translations, called a **translation-lookaside buffer**.

Because we access the TLB instead of the page table on every reference, it must include other status bits, such as the dirty and reference bits, just like other caches.



TLB Table Details

- Valid Bit: is the data valid for this entry?
- Dirty Bit: do we need to write back when this it is replaced?
- Reference Bit: approximates LRU as before.
- Tag Field: the entry is referenced by VA.
- Physical Page address: records the corresponding PA.

```
typedef long long int VA;
typedef long long int PA;
typedef long long int VPN;
typedef long long int PPN;
typedef long long int OFFSET;

VPN getVPN(VA);                                 /* Extracts the first 52 bits of a VA */
OFFSET getOffset(VA);                           /* Extracts the last 12 bits of a VA */
PA concat(PPN, OFFSET);                         /* Concatenate PPN and OFFSET to produce a PA */

class Page {};                                  /* Page: a block in virtual memory system */
class Data {};                                  /* Data: 32-bit data stored inside RAM index by a physical address */

class _TLB {                                    /* Translation Lookaside Buffer */
    struct TLBRow { int validBit; int dirtyBit; int refBit; VPN tag; PPN ppn; };
    vector<TLBRow> table;
public:
    bool hit(VPN vpn);                          /* Returns true if given vpn exists in TLB */
    PPN get(VPN vpn);                           /* Returns the corresponding PPN of the given VPN */
    void add(VPN vpn, PPN ppn);                 /* Adds a new entry to the table, removes entry if full */
};

class _PageTable {                              /* Page Table inside RAM */
    struct PageTableEntry { int validBit; int dirtyBit; int refBit; PPN ppn; };
    vector<PageTableEntry> table;
public:
    bool hit(VPN vpn);                          /* Returns true if given vpn exists in TLB */
    PPN get(VPN vpn);                           /* Returns the corresponding PPN of the given VPN */
    void add(VPN vpn, PPN ppn);                 /* Adds a new entry to the table, removes entry if full */
};

struct _RAM {
    void load(Page page);                       /* Load a page into RAM */
    PPN locate(Page page);                      /* Locates PPN on RAM */
    Data getData(PA pa);                        /* Returns M[pa], the data stored at physical address pa */
}

struct _DISK {
    Page get(VPN vpn);                          /* Extract page given virtual page number */
}

struct _Cache {
    bool hit(PA pa);                            /* Returns true if given pa is cached */
    Data get(PA pa);                            /* Extract data given physical address */
}

/* Globals */
_TLB TLB;                                       /* Our TLB */
_PageTable PageTable;                           /* Our PageTable */
_RAM RAM;                                       /* Our RAM */
_DISK DISK;                                     /* Our Disk */
_Cache CACHE;                                   /* Our Cache */

/* Given a virtual address, return the corresponding physical address */
PA translation (VA va) {                        /* va.length() = 64; */

    VPN vpn = getVPN(va);                       /* vpn.length() = 52; */
    OFFSET offset = getOffset(va);              /* offset.length() = 12; */
    PPN ppn;                                    /* ppn.length() = 20 */
    PA pa;                                      /* pa.length() = 32 */

    if (TLB.hit(vpn)) {                         /* If TLB hits */
        ppn = TLB.get(vpn);                     /* Get the corresponding PPN */
    }
    else {                                      /* If TLB misses */

        if (PageTable.hit(vpn)) {               /* If Page Table hits */
            ppn = PageTable.get(vpn);           /* Get the corresponding PPN */
            TLB.add(vpn, ppn);                  /* Updates TLB, removes entry if necessary */
        }
        else {                                  /* If Page Table misses */
            Page p = DISK.get(vpn);             /* Find page in disk using vpn */
            RAM.load(p);                        /* Load page into RAM */
            ppn = RAM.locate(p);                /* Locate the recently-loaded page on RAM */
            PageTable.add(vpn, ppn);            /* Update page table, removes entry if necessary */
            TLB.add(vpn, ppn);                  /* Update TLB, removes entry if necessary */
        }
    }
    pa = concatenate(ppn,offset);               /* Finally, we get the physical address */
    return pa;                                  /* Return the physical address we got */
}

/* Given a physical address, find its data */
Data getData(PA pa) {
    Data result;                                /* Our output */

    if (Cache.hit(pa)) {                        /* If PA is cached */
        result = Cache.get(pa);                 /* Extract the cached data */
    }
    else {                                      /* If PA is not cached */
        result = RAM.getData();                 /* Load the data from RAM */
        cache.add(pa, result);                  /* Update Cache, remove entry if necessary */
    }
    return result;                              /* Return the data we got */
}
```

C-Style Pseudocode for address translation and cache access