3 The Memory System

3.1 Memory Hierarchy

Types of Memory

- storage medium (physical types)
  - Semiconductor（半导体介质）: RAM & ROM; *fast, low-power.*
  - Magnetic（磁介质）: Disk & Tape; *low, large-capacity, large-size.*
  - Optical（光介质）: CD & DVD. *cheap, easy-to-store, easy-to-carry.*
- storage mean
  - **random access memory**: Each memory unit can be accessed (read and write), and the access latency doesn't depend on the unit's physical location. (*semiconductor*);
  - **sequential access memory**: Memory units can be accessed sequentially, and the access latency depends on the unit's physical latency. (*disk, tape, CD & DVD, etc.*).
- volatility of memory
  - Volatile Memory（易失）: lost data when power is off. (*RAM: DRAM, SRAM*);
  - Non-Volatile Memory（非易失）: keep data when power is off. (*magnetic-memory, optical-memory, phase-change memory,NVRAM*)
- role in the computer
  - CPU register;
  - Cache (SRAM);
  - Main memory (DRAM);
  - Solid Storage Disk (SSD);
  - Disk, CD & DVD.

Memory Hierarchy

- (small, fast, expensive) register; cache; main memory; SSD; disk (big, slow, cheap).
- Bigger (capacity) is slower.
- Faster is more expensive.

> **The Trend of DRAM Technology**
>
> Capacity: 4x in 3 years;
>
> Latency: 2x in 10 years.
>
> The speed of DRAM is increasing slowly when comparing with CPU (2x in 1.5 years according to *Moore's Law*), which makes the gap between them. How to overcome the gap? The memory hierarchy and the locality principle.

The Locality Principle

The program prefers to use the data and instructions in local address, which the program has accessed previously.

- Temporal locality（时间局部性）;
- Spatial locality（空间局部性）.

> [*Example*]

```
sum = 0;
for (i = 0; i < n; i ++)
    sum += a[i];
return sum;
```

- Access the array sequentially: spatial locality;
- Access `i` and `sum` every loop: temporal locality;
- Read the instruction sequentially: spatial locality;
- Repeat execute the instructions in the loop: temporal locality.

How is hierarchy managed

- registers with memory: by compiler (programmer);
- cache with memory: by the cache controller hardware.
- main memory with disks:
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by programmer.

3.2 Static RAM

Features

- 6 transistors per bit of SRAM.
- Optimized for speed (first) and density (second)
- Static: the data remains until the power is off.
- Expensive;
- Fast;
- High power consumption.

The structure of SRAM: See the notes of Chap 2 in EI209.

The memory chip's structure: See the notes of Chap 2 in EI209.

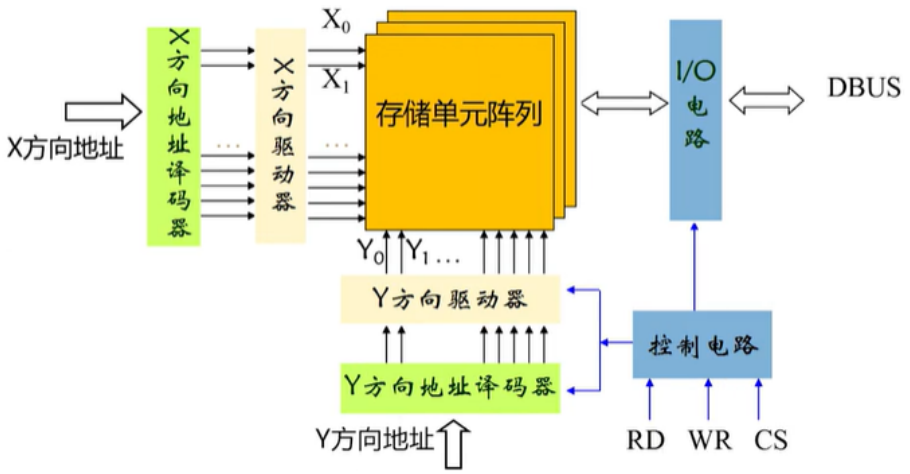
How to access the memory array?

- **Double decoding**: See the notes of Chap 2 in EI209. (*the common method*)

  If the address is n-bit, then the total capacity is 2^n, and the total decoder lines are $2 \times 2^{n/2} = 2^{n/2+1}$.

- **Single decoding**: Only use a decoder to access the memory.

  If the address is n-bit, then the total capacity is 2^n and the total decoder lines are 2^n.



3.3 Dynamic RAM

Features

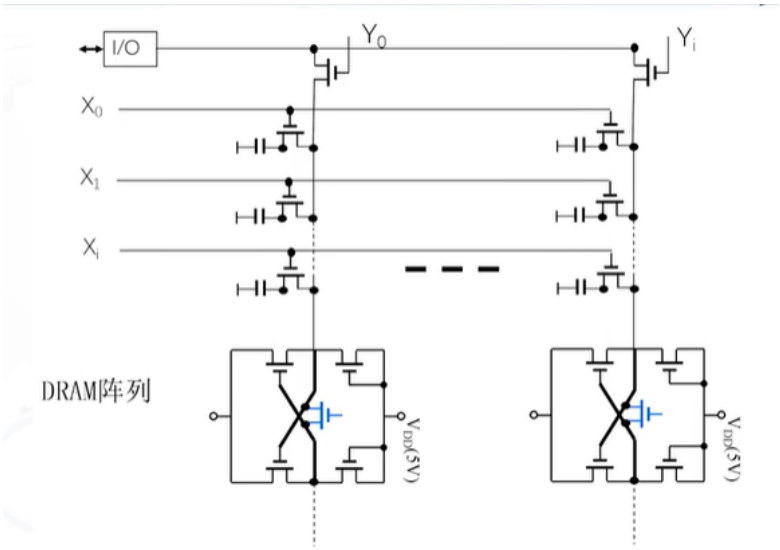
- An transistor and a capacitor per bit of DRAM.
- Optimized for density (in terms of cost per bit).
- Use electric charge to store the information.
- Dynamic: the memory should be '**refreshed**' often.
- Less expensive than SRAM;
- Slower than SRAM;
- Low power consumption comparing to SRAM;
- Reads destructive.

Refresh（刷新）: to replenish the charge in DRAM.

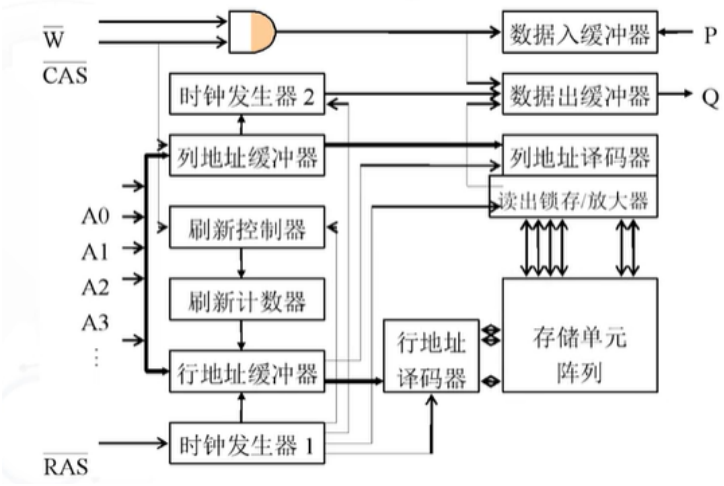
- *The cycle of refreshing*（刷新周期）: the time period between this refreshing operation and last refreshing operation.
- *Maximum cycle of refreshing*（最大刷新周期）: the time period between storing data and leaking all the charge.
- Use **DRAM refresh amplifier**（刷新放大器）to refresh the data in DRAM.

How to refresh DRAM: refresh by row.

- **Buffer (or latch)**（缓冲区，锁存器）: there is a row of buffer in DRAM arrays, can be used to store the whole row's data when reading or writing DRAM. Also, when refreshing, the data goes to the buffer first and then go back to the DRAM. (*opening* and *closing* a row)
    - *Row buffer acts as a cache within DRAM.*



- Asynchronous（异步刷新）: the refreshing of every line is distributed in the whole cycle. Therefore, the total times of refreshing is depend on the total line number of the DRAM array.
- A refreshing operation includes two access operations: read & write.



DRAM has extra row address buffers, column address buffers and $\overline{RAS}, \overline{CAS}$, because the capacity of DRAM is larger than SRAM, so the number of addressable units is bigger. Suppose DRAM supports n-bit address, it only need $n/2$ lines because the address will be separated and delivered in 2 times.

- **RAS** (Row Access Strobe, 行地址（译码）触发)
- **CAS** (Column Access Strobe, 列地址（译码）触发)
- When *RAS* is in low voltage, it means it's the row address; when *CAS* is in low voltage, it means that it's the column address.

The reading cycle

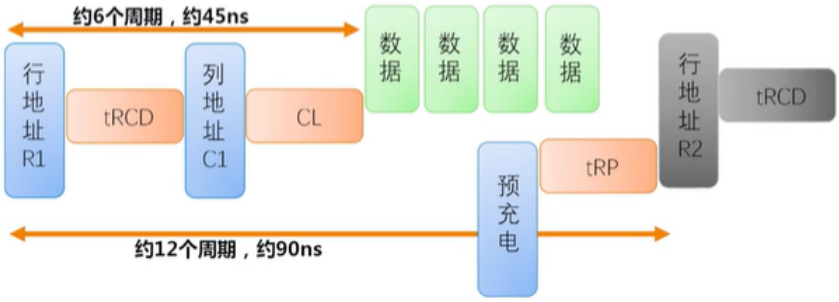
- Read the row address from address line, set *RAS* in low voltage: the data go to the buffer. (*[Precharge and] Row Access*)
- Read the column address from address line, set *CAS* in low voltage: read data from buffer. (*Column Access*)
- Output the data, and refresh the whole row's data. (*Data Transfer*)

Cost of Accessing DRAM

- Row buffers hit: ~20 ns access time, only transfer data;
- Empty row buffers access: ~40 ns access time, read data from arrays and transfer data;
- Row buffers conflicts: ~60 ns, write data back, read data from arrays and transfer data.
- **Two main methods**: write back after every access & let the data stay in row buffer.

Synchronous DRAM (SDRAM)（同步DRAM）

- **Burst**（猝发传输）: transfer the data of the continuous address in one time. *CAS* represents the beginning of the burst process.
- **Metrics**



  - **tRCD**: Row to Column Delay; the delay time (cycle) from RAS to CAS.
  - **tCL**: CAS Latency: the delay time (cycle) from CAS to the output.
  - **tRP**: RAS Pre-charge: the delay time (cycle) of closing the row (*If a part of the data want to read is in the next row*).

- **Worst cycle time**: $tRCD + tCL + tRP$;

Double Data Rate SDRAM (DDR-SDRAM)（双倍速率SDRAM）

The current main technology. It can perform *two transfers* in a clock cycle.

> [*Example*] Names of DDR SDRAM.
>
> "DDR x" means 2^x bits every burst transfer.
>
> - DDR 2: low power (1.8V), high clock rates (400 MHz);
> - DDR 3: 1.5V, 800 MHz;
> - DDR 4: 1-1.2V, 1600 MHz.
>
> Graphics Memory
>
> - GDDR5 is graphics memory based on DDR3.
> - Achieve 2-5x bandwidth per DRAM vs DDR3: Wider interfaces; higher clock rate.

3.4 Constitution of Memory

Performance Metrics

- Latency（延迟）: the time it needs to access a byte.
  - Access time（访问时间）: the time period between a request and a data-receiving.
  - Cycle time（周期时间）: the time period between two requests.
  - Usually, cycle time > access time.
- Bandwidth（带宽）: the amount of data transferred from memory to CPU in a unit of time.
  - Bandwidth = the width of data bus * the transfer rate of data bus.

Key Factors of Latency of Main Memory

- The transfer time between CPU and *memory controller*（内存控制器）;
- The latency of memory controller:
  - The instructions are in queue (scheduling latency);
  - The instructions are transformed to the certain instructions in the memory controller.
- The transfer time between memory controller and DRAM;
- DRAM Bank latency（存储芯片体延迟）;
- The transfer time between DRAM and CPU (through memory controller).

Current Trend: The memory controller is integrated in CPU.

> [*Example*] The *burst* in SDRAM can reduce the latency of memory, because it can transfer more data in a time and as a result, the times need for transferring is less.

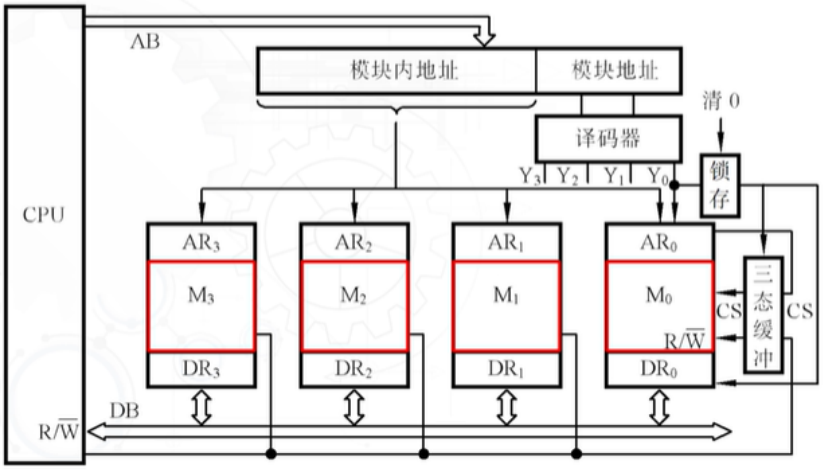
Optimization

- Fast Page Mode Operation: multiple accesses to same row; it enables multiple reads from page without RAS.
- Synchronous DRAM: added clock to DRAM interface; burst mode with critical word first.
- Double Data Rate (DDR): DDR DRAM transfers data on both rising and falling edge of the clock.
- Wider interfaces: more complex implementation, and may cause conflict problems.
- Multiple banks on each DRAM device.

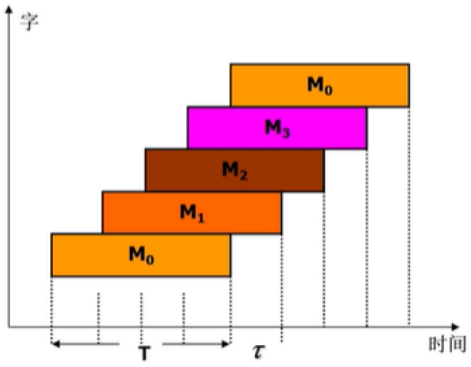
Multi-bank（多体交叉）: It's a little bit different from *bit-extension*. Bit-extension extend memory word width, but multi-bank do not extend it, when the data bus width is greater than the memory word width (usually 8-bit), then we can transfer more words at one time, so we need multi-bank to make fully use of the data bus.

Notes: 多体交叉如果是 8 个 1-bit 交叉得到 8-bit（也就是 1 memory-word），那么实际上和 bit-extension 没有区别。

- *high-bit multi-bank*（高位交叉） is extensible (continuous words stored in same bank), but *low-bit multi-bank*（低位交叉）can be performed *parallel*. We introduce *low-bit multi-bank* here.



- The continuous words are stored in different banks (e.g. 0-M0, 1-M1, 2-M2, 3-M3, 4-M0, 5-M1, 6-M2, 7-M3, 8-M0, ...) (*low-bit multi-bank*)
- Use the last two bits to locate the target.
- The access process of multi-bank memory chip.



  - T is the access cycle time of a bank.
  - τ is the transfer cycle time of data bus.
  - m is the number of banks. (also called degree of memory interleaving（交叉存取度）)

$$T \leq m\tau \quad \Longrightarrow \quad m \geq \frac{T}{\tau}$$

  The total time of access n bytes is reduced.

  - If $m = n$, we have $t_1 = T + (n-1)\tau$;
  - If $m = 1$ (simple), we have $t_2 = nT$.
  - $t_1 < t_2$, so multi-bank can reduce the average access *latency*, and also increase the *bandwidth*.

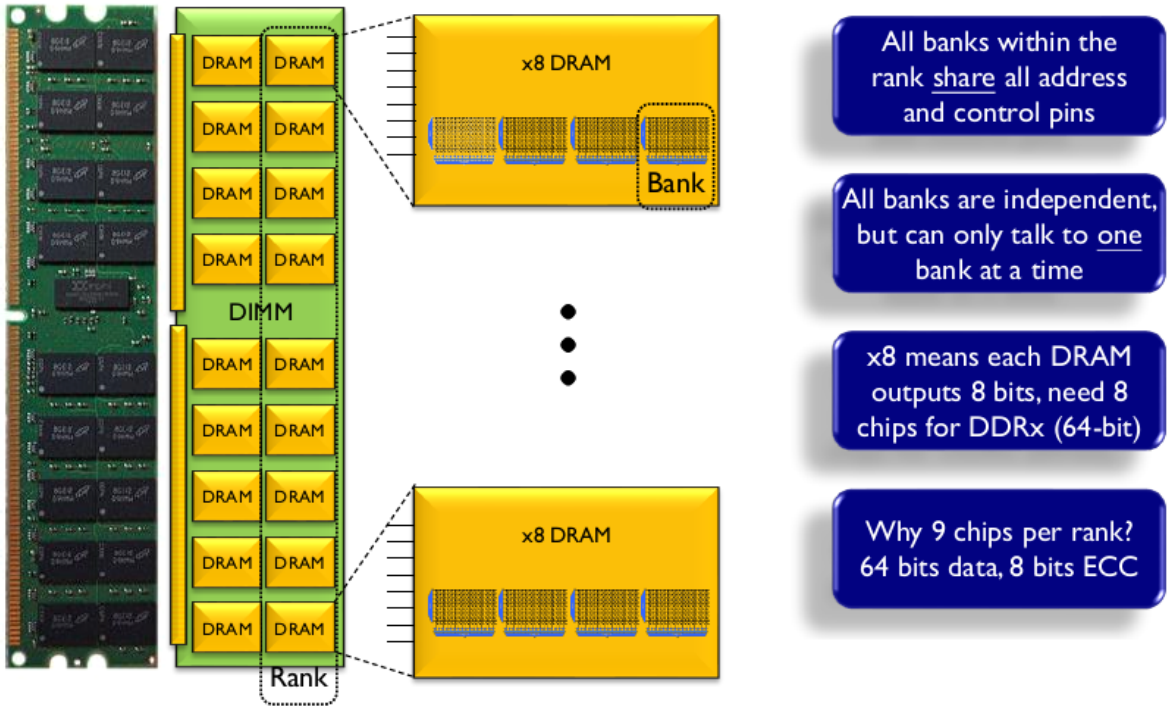
DRAM chips

- DDR SDRAM: 2 banks; DDR5 SDRAM: 32 banks.

- Many chips together can extension the word width and the capacity of memory.

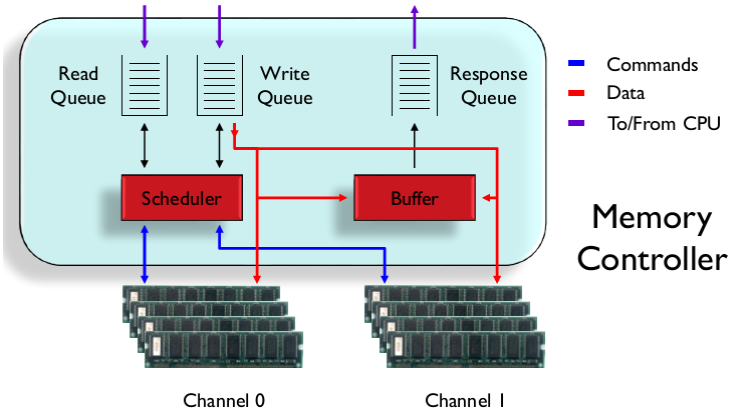
Memory Extension: See the notes of Chap 2 in EI209.

DRAM Organization



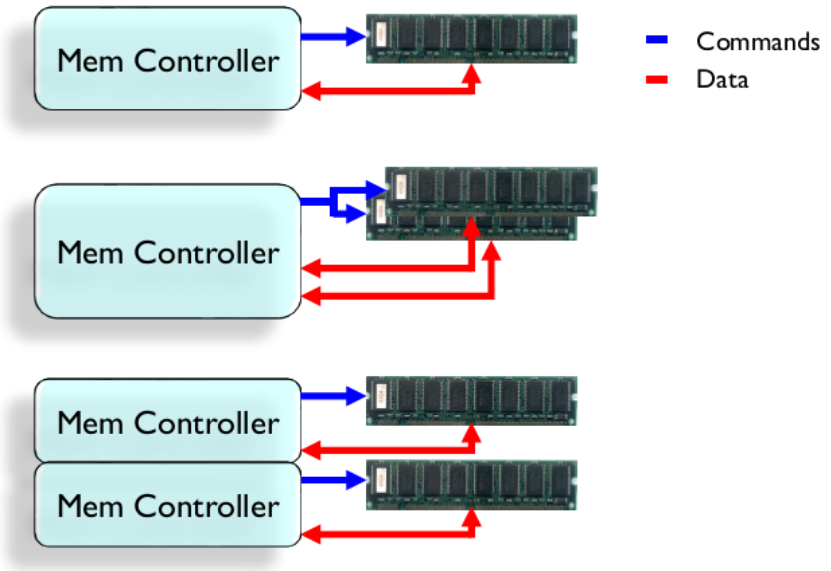
- DIMM: Dual-Inline Memory Module
- Every DRAM can output a 8-bit data (a word). If we want to get 8 words a time (64-bit machine), we need 8 DRAM and 1 extra DRAM for validation（验证，ECC）. (total 9 DRAM). So there are 9 DRAM in a *Rank*.
- The DRAM in Rank is like the Bank in DRAM (*multi-bank - like*).

Memory Controller



- Memory controller connects CPU and DRAM;
- Memory controller receives requests after cache misses (possibly originating from multiple cores).
- Complicated piece of hardware, handles:
  - DRAM refresh;
  - Row-buffer management policies;
  - Address mapping schemes;
  - Request scheduling.

Memory channel



We can use multiple channel for more bandwidth.

PHYSICAL Location of Data in Memory

- Channel ID (Which DIMM/Channel?)
- Rank ID (which rank?)
- Bank ID (which bank?)
- Row ID (which row?)
- Column ID (which column?)

Overcoming memory latency

- Caching
  - reduce average latency by avoiding DRAM altogether;
  - Limited capacity and may have compulsory misses.
- Prefetching: guess what will be accessed next and put it into the cache.
- Memory-level parallelism: perform multiple concurrent accesses.

3.5 Cache

Cache, which is based on SRAM, is a low-capacity and high-speed memory near the processor, and it is automatically controlled by hardware.

- Store the frequently-accessed data in cache;
- CPU search data in cache first, then the main memory;
- We hope that most of the data we need to access is in cache.

Block: the transfer between main memory and cache has a unit of block. Usually the size of block is greater than a word (e.g. 8 words).

Hit: The data we want to access is in cache.

Miss: The data we want to access is not in cache. Then we fetch the data we need and put it in cache. Placement policy（替换策略） decides which block to be replaced, and write policy（写入策略） decides which block to place the data.

- **Compulsory misses** (cold misses): when a block is accessed for the first time;
- **Capacity misses**: when a block is not in the cache because it was evicted because the cache was full;

- **Conflict misses**: when a block is not in the cache because it was evicted because the cache *set* was full (only happens in *direct-mapped* or *set-associative* caches);

> **Rule of thumb**: miss rates change in proportion to $\sqrt{\text{size of cache}}$.

Hit Accuracy: the probability of hit in all access operations.

- N_C: Access data in cache;
- N_m: Access data in main memory;
- h: the hit accuracy of cache.

$$h = \frac{N_c}{N_c + N_m}$$

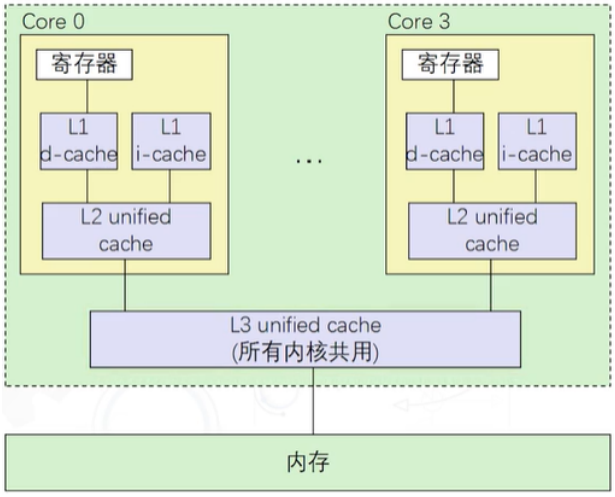
- t_c: the access time when data is in cache;
- t_m: the access time when data is in main memory;
- t_a: average access time.

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

How to reduce average access time

- reduce the hit time;
  - (*main*) Smaller & simple cache; direct-mapped cache; smaller blocks.
  - Use write buffer.
- reduce the miss time;
  - Smaller blocks (*not practical*);
  - Use write buffer & victim cache.
  - For large blocks fetch:
    - **Early Restart**: request words in normal order, and sends missed word to the processor as soon as it arrives. Do not need to wait until the whole block is stored in cache.
    - **Critical Word First**（关键字优先）: request missed word from memory first, and send it to the processor as soon as it arrives.
  - **Giving priority to reads over writes**（读权限比写权限高）
    - Place write misses in a write buffer, and let read misses overtake writes;
    - Flush the writes from the write buffer when pipeline is idle or when buffer full;
    - Reads to the memory address of a pending write in the buffer now become hits in the buffer.
    - （可以理解为，先将待写数据写入 write buffer；如果下一条指令是读，则可以直接从包括 write buffer 在内的 cache 读取，不用等待写 memory 指令结束）
  - **Non-blocking Cache**: Cache can service multiple hits while *waiting on a miss*.
  - Use **multiple cache levels**.
    - Upper components emphasize t_c (hit time);
    - Lower components emphasize h (hit rate).
- improve the hit accuracy.
  - Bigger cache; more flexible placement (increase associativity); larger blocks (reduce compulsory miss).
  - **Victim cache**: small buffer holding most recently discarded blocks from cache.
  - **Prefetching**（预取）
    - *hardware prefetching*（硬件预取）:

        - fetch two blocks on a miss, the requested block i and the next consecutive block $(i+1)$.
        - **Stride prefetch**: if observe sequence of accesses to block $b, b+N, b+2N$, then prefetch $b+3N$ etc.
    - *software prefetching*（软件预取）: compiler inserts instructions at proper places in the code to trigger prefetches. It requires ISA support (nonbinding prefetch instruction)
- **Compiler optimizations**: improve time & space locality. (e.g. *merging arrays* and *loop fusions*).

> [*Example*] Intel i7 processor
>
> 
>
>
> *Block size*: 64 Bytes;
>
> *L1 i-cache & d-cache*: 32 KB, 4 cycles per access;
>
> *L2 unified cache*: 256 KB, 10 cycles per access;
>
> *L3 unified cache*: 8 MB, 40-75 cycles per access

Key problems

- Data Identification（数据查找）.
- Address Mapping（地址映射）.
- Placement Policy（替换策略）.
- Write Policy（写入策略）.

3.6 The Address Mapping of Cache

The address in main memory can be viewed as the block number in memory and the address in block; **The address in cache** can be viewed as the block number in cache and the address in block.

Address Mapping can map the the block number in memory to the block number in cache that have the same data (hit) or nothing (miss). If we miss the data, then we put the block which contains the data into the cache (may need replacement, using *placement policy*).

Three mapping methods:

- direct mapped（直接映射）;
- fully-associated（全相连）;
- set-associated（组相连）.

Direct mapped（直接映射）

Memory block number j, total number of cache n, then the destination block number in cache $i = j \bmod n$.

Divide the main memory address into 3 parts: district number（区号） + block number in cache（cache中的块号） + address in block（字地址）.

Divide cache address into 2 parts: block number（块号）+ address in block（字地址）

Use address in block to locate the data in a block.

Every block in cache has a *tag*, which is the *(memory) district number* of the data stored in this block. Comparing the *tag* with the *district number* of the access data can check whether the process hits the data.

If miss, access the main memory to get the data and use it to replace the data in the target block directly.

- *Advantage*: quick transformation speed, one-to-one mapping, easy to implement, simple substitution algorithm.
- *Disadvantage*: easy to hit, low usage of cache, low hit rate. (*ping-pong effect*)

Fully-associated（全相连）

Divide the access address into 2 parts: the main memory block number, the address in block. Use address in block to locate the data in a block. Search the main memory block number in cache.

If miss, access the main memory to get the data.

Every block in cache also has a *tag*, which is the *block number* of the data in main memory. Comparing the *tag* with the *block number* of the access data can check whether the process hits the data.

Determine which block should be replaced (*placement policy*), and replace it with the data we get. (We will discuss the placement policy in Chap 3.7)

- *Advantage*: do not conflict until the cache is full, low conflict rate, high usage of cache.
- *Disadvantage*: one-to-many mapping, complex substitution algorithm, slow searching process.

Address Mapping: use part of the content of storage as the address.

Set-associated（组相连）

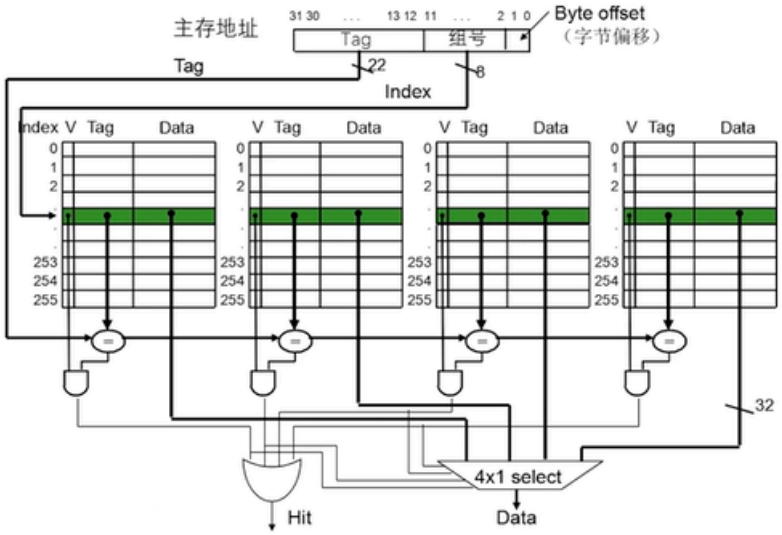
Special case: direct mapped & fully-associated.

Ignore the address in block, and the last several bits of the main memory address is the set number of the main memory block. Ignore the address in block, and the last several bits of cache address is the set number of the cache block.

Use the high bits of the cache block (*way*) to distinguish the different blocks in the same set.

Thus, the main memory address can be divided into 3 parts: district number + set number + address in block; the cache address can be divided into 3 parts: way number + set number + address in block.

The way number equals to the size of each set.



- Use the set number to locate the set number in cache;
- Compare the district number (*tag*) with *tag* in cache; If found, then hit and output the data. If not found, then miss, and use the *substitution policy* of full-associated to replace the data.

Given a cache,

- Increase the length of tag (way number, associativity) , then the set of number will decrease. (*Special case: fully-associated*)
- Increase the number of set, then the length of tag (way number, associativity) will decrease. (*Special case: direct mapped*)

Set-associated is between fully-associated and direct mapped.

- rather high hit rate, rather fast in speed, complex in implementation.

Strategy

- Low capacity: *set-associated* or *fully-associated*, high hit rate;
- High capacity: *direct mapped*, fast but low hit rate.
- Require high speed: *direct mapped*, fast;
- Do not require high speed: *set-associated* or *fully-associated*, high hit rate but slow.
- The optimal choice is a compromise, it should consider access characteristics, technology and cost. Simplicity often wins.

3.7 The Placement Policy of Cache

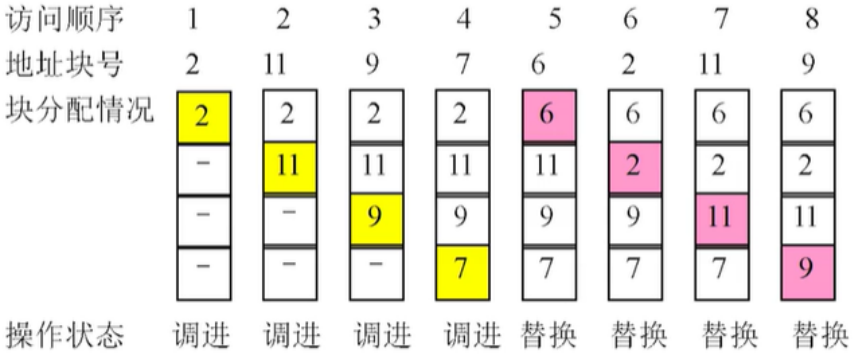
Random Replacement（随机替换）: randomly replace one of the data in the same set.

FIFO (First In First Out)（先入先出）: replace the earliest element in the same set.

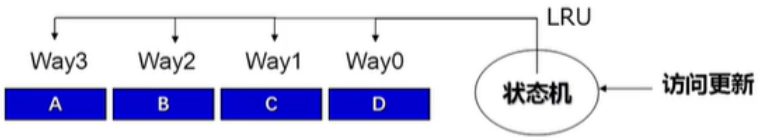
LFU (Least Frequently Used)（近期最少使用）: replace the least frequently used element in the same set.

LRU (Least Recently Used)（近期最久未使用）: replace the least recently used element in the same set.

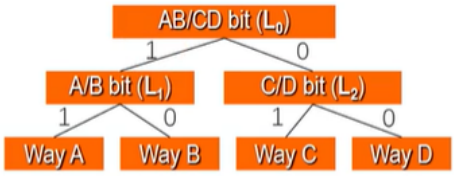
- **Bumpy phenomenon**（颠簸现象）: We use the new block A to substitute the original block B, and the choice may be bad because block B will soon be used (of course we don't know when we are doing the substitution). Then, when block B is used, we will use it to substitute the block C, and the choice may be bad again because block C will soon be used (of course, we don't know before either). And repeat the process. It may need many substitutions and the hit rate of cache will be extremely low.



- **Implementation in hardware**



  - LRU needs extra hardware bits to store the state of each block: the access order of total n blocks, and n is the way number（路数） of a cache. There are $n!$ situations totally, so it needs $\log n!$ extra bits, that is $O(n \log n)$ extra bits.
  - LRU needs extra time to update the state of each block, so it increases the access time of cache.
  - LRU method needs $\log n$ extra bits every block.
- **Pseudo LRU**（伪LRU）: an binary-tree-based algorithm, only needs $O(n)$ extra bits.
  - The binary tree is used to store the historical access order. (left-subtree: 1, right-subtree: 0)



  - We only use L_0, L_1, L_2 three extra bits to record the state.
  - When accessing and hit, we use the updating algorithm. Referring to *updating table* and change the state of three bits. (e.g. when access way B and hit, we set $L_2L_1L_0$ to L_201)
  - When accessing and miss, we use the substitution algorithm. Referring to *substitution table* and make a substitution. (e.g. the current state is $L_2L_1L_0 = 001$, then we will substitute block C)
  - *Less hardware usage, faster speed* than LRU.

3.8 The Write Policy of Cache

Write Policy: to keep the synchronization of data between memory and cache.

EVERY write policy will need an extra bit: valid bit

Write Policy when Cache Hits

- **Write-through**（写直达法）: write data to both the memory and cache directly. See here: http://www.icsa.inf.ed.ac.uk/research/groups/hase/models/coherence/wti-model-t.html.
  - Low speed; generate more traffic; but keep synchronized.

- **Write-back**（写回法）: only write data to the cache, and delay the write process to the memory. When the block in cache will be replaced by another block, then write the block data to the memory.
    - This method has an influence to the placement policy: when we are going to substitute block A with block B, we will check block A.
        - No modification: no need to write back, discard the block;
        - Have modifications: need to write back to memory first.
    - Add an extra **dirty bit** in front of every block of cache to record whether the block has been modified. (*0 - no, 1 - yes*).
    - An cache *line* contains: valid bit, dirty bit (maybe), LRU bits (maybe), tag bits, total data block bits.
    - High speed; generate less traffic; sometimes not synchronized.

Write Policy when Cache misses

- **Write-allocate**（按写分配法）: read the data into the cache, and update the data only in cache.
    - A better method when it needs to access the same place continuously;
    - Along with **Write-back**.
    - Use the high-speed feature of cache, reduce the number of memory writing.
- **No-write-allocate**（不按写分配法）: write the data to the memory directly, do not use cache.
    - Have no effect on the data in cache
    - Along with **Write-through**.
    - Securer because of the data synchronization.

How to choose?

- Security, synchronization: **write-through**, **no-write-allocate**;
- Speed, efficiency: **write-back**, **write-allocate**.

3.9 Multi-level Cache

Inclusive caches:

- Lower-level cache has a copy of every block in higher-level caches;
- Wastes capacity of lower-level caches;
- Simplifies finding a cache block by another entity (e.g. other processors).

Exclusive caches

- A block may reside in only one level of the cache hierarchy;
- Maximize aggregate capacity of the cache hierarchy.
- Requires a uniform block size for all cache levels.

3.10 Cache-friendly Code

> [*Example*] In C programming language, the array is allocated by line with a continuous memory in main memory.
>
> ```
> for (i = 0; i < N; i ++)
> sum += a[0][i];
> ```

```
The code above is a cache-friendly code because it uses the space-locality.
Then the cache miss rate = the size of array / the size of cache block.

- ```c
for (i = 0; i < N; i ++)
    sum += a[i][0]
```

The code above is not a cache-friendly code because it doesn't use the space-locality. Then the cache miss rate = 100%.

How to write a cache-friendly code?

- Make the common case run faster: focus on the core of the code (often in an inner loop).
- Minimize the cache miss rate of the code.
  - Access contiguous variables (space-locality);
  - Quote the same variable frequently (time-locality).

[*Example*] The matrix multiplication.

```c
// code 1
for (int i = 0; i < n; ++ i)
    for (int j = 0; j < n; ++ j) {
        int sum = 0;
        for (int k = 0; k < n; ++ k)
         sum += a[i][k] * b[k][j];
     c[i][j] = sum;
 }
```

```c
// code 2
for (int k = 0; k < n; ++ k)
    for (int i = 0; i < n; ++ i) {
        int r = a[i][k];
        for (int j = 0; j < n; ++ j)
            c[i][j] += r * b[k][j];
    }
```

Code 1 and code 2 do the same thing, but code 2 is more cache-friendly because it access the elements *by line*.

Change the loop order may make the code cache-friendly.

Also, divide the matrix into blocks, so that blocks can be stored in cache. Do the matrix multiplication to the blocks and it will be much faster (the blocks are in the cache!). Therefore, we can make more optimizations to the code.

3.11 Virtual Memory

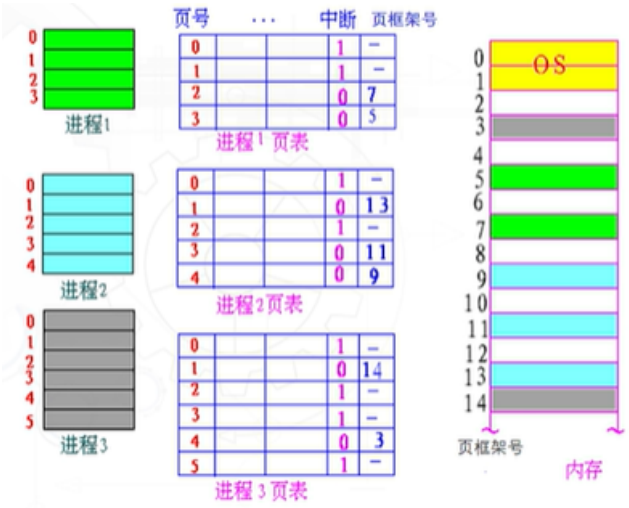
The program's code is greater than the memory. The memory cannot keep storing the code during the execution of the program. Multitasking will also cause the problem. Thus, we need **virtual memory**.

- Only store the current part of the program in the memory, and keep others in the disk. Dynamically substitute the program between memory and disk.

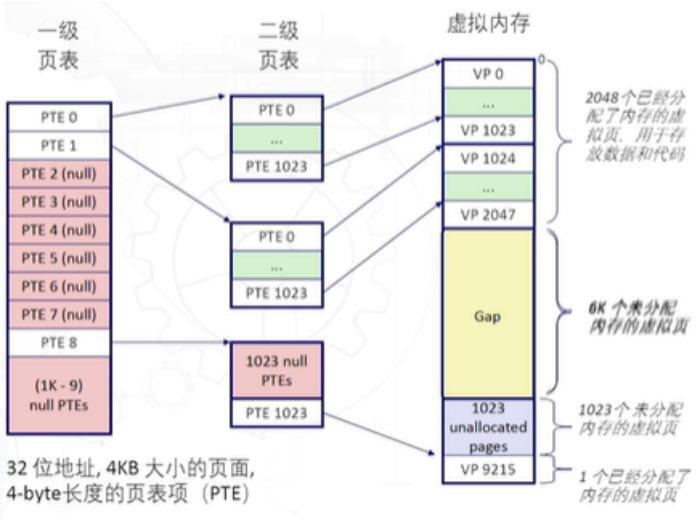
Page（页）: a program's address space can divide into several pages with the same size (usually 4KB).

- We store some pages in the memory while some pages are still in the disk.
- If we divide the memory into several parts according to users（分段）, we will need to re-organize the memory frequently, since there will be many patches（碎片）.
- Store some pages into the memory and then the program can execute. When the program needs other pages, it will send an interrupt（缺页中断请求）. When the operating system deal with the interrupt, it will fetch the required pages into the memory so that the program can continue executing.
- To implement these, the system need to have:
  - Page table（页表机制）;
  - Address transformation（地址变换机构）;
  - Missing-page interrupt（缺页中断机构）;
  - Page replacement algorithm（页面置换算法）.

Page Table（页表）

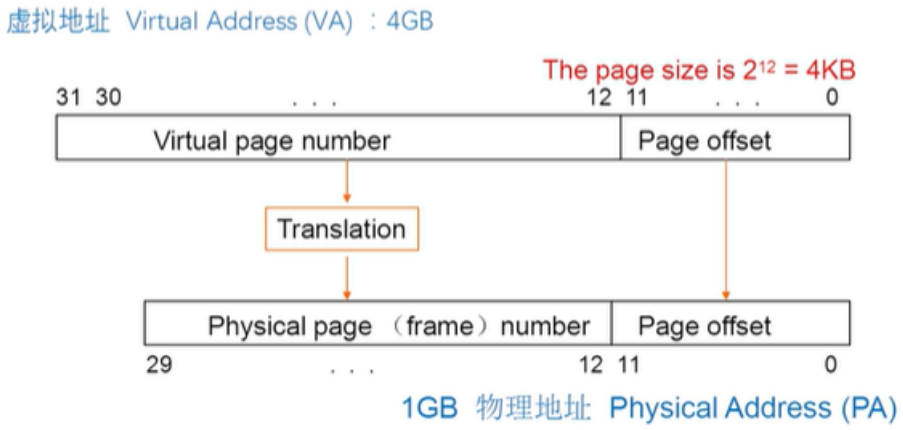


- Page table contains:
  - Page number, page framework number;
  - Status state P: whether the page is in memory. It is used when program access the page.
  - Access times A: the access times of the pages. It is used when substitutions are needed.
  - Modification state M: whether the page is modified in memory. It is used when substitutions are needed.
  - External-memory address: the page's address in the external memory. It is used when the page needs to be stored in the memory.
- The page table's length is proportional to the address number（地址数目） and the process number（进程数目）, so it needs lots of space. Therefore, the page table cannot be stored in register. The page table is stored in the *main memory*.
  - *Linear page table* is very large and usually it cannot be stored in the main memory.
  - **Multi-level page table**（多级页表）: combine the page tables of highest level together and we can get the real page table. It will save a lot of space, since usually we don't use every space in the virtual memory, so many empty page tables do not have to be stored.

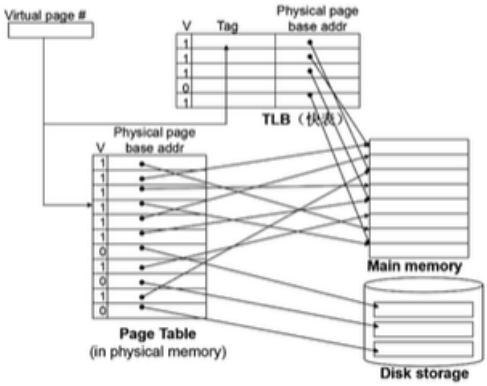


- **Inverted page table**（反向页表）: use a physical-address to virtual-address mapping, with the help of hash algorithm.

Address Transformation（地址转换策略）



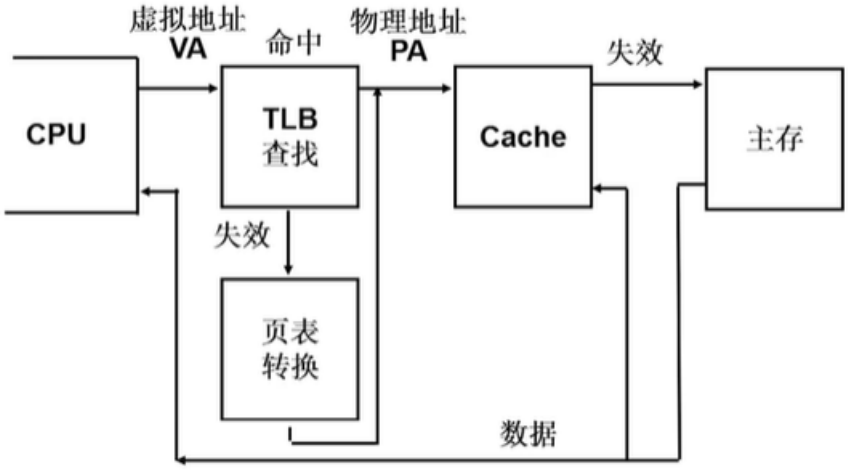
- Given a virtual address of the memory unit, we need to transform it into the physical address of the memory.
  - Page offset doesn't change.
  - Translate the virtual page number to the physical page number.
  - When the virtual page is not in the memory, will cause **page fault**（页面失效）.
- **TLB (Translation Lookaside Buffers)**（快表）



Since the page table is stored in the *main memory*, if we refer to the page table in translation process, it will be much slower. Then we have TLB. Store the frequently-visited page's address information in cache. Therefore, we can refer to TLB first before we refer to the page table in main memory.

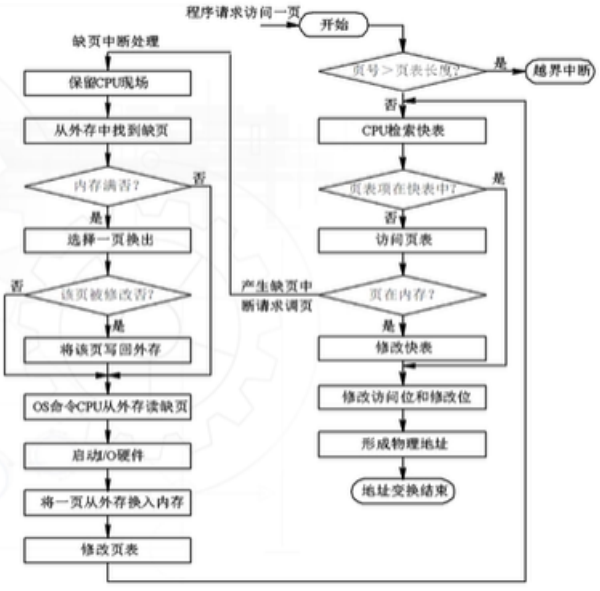
- Fully-associative;
- Set-associative;
- Direct mapped.

A TLB line is smaller than a cache line. Therefore its speed is faster than cache. TLB is like a bridge between CPU and cache.



- **TLB miss**: the page is in the memory, but not in TLB. Fetch the information from the main memory and update the TLB. It usually takes 10 CPU cycles.
- **Page fault**: the page is not in the memory. Fetch the page from the *disk* and update the page table in memory, and update TLB. It usually takes 10^6 CPU cycles.

Thus, **TLB miss** $\rightarrow$ access **Page table**; **Page table miss** $\rightarrow$ fetch page from *disk*. This is the process to get the physical address.



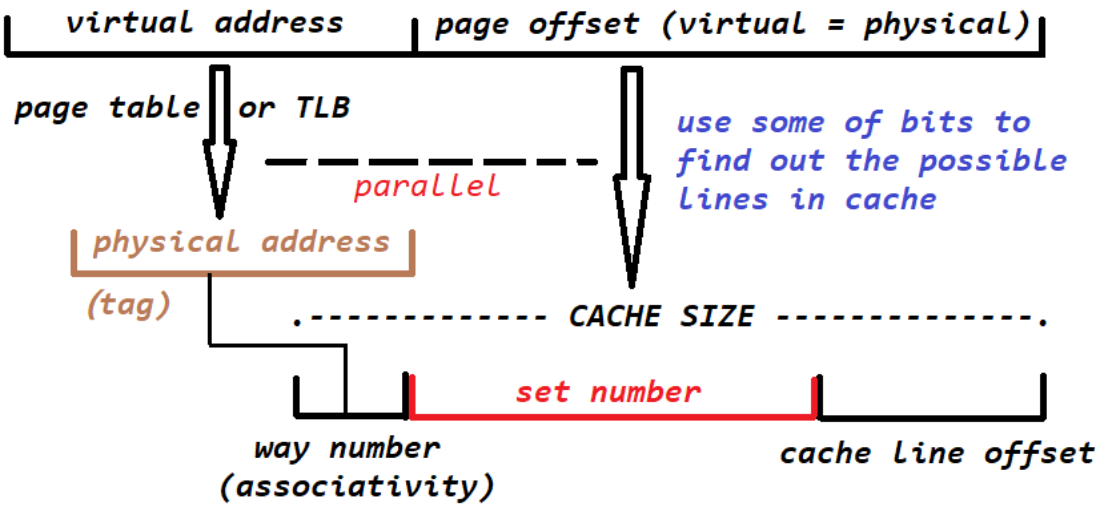
Address a cache in virtual memory system

- **Physical-addressed caches, PI-PT**: perform address translation before cache access. Hit time is increased to accommodate translation, high latency.
- **Virtually-address caches, VI-VT**: perform address translation after cache access if miss. Hit time does not include translation. But *aliases* may happen.
- **Virtually-indexed Physically-Tagged, VI-PT**: perform address translation and searching the *offset* in cache parallelly!

  Need to guarantee the following equation.

$$\text{cache\_size} \leq \text{page\_size} \cdot \text{associativity}$$

Why?



Actually, we need $\text{set\_number} \times \text{cache\_size} \leq \text{page\_size}$. Because of the fact that $\text{set\_number} \times \text{cache\_size} \times \text{associativity} = \text{cache\_size}$, we can derive the formula above.