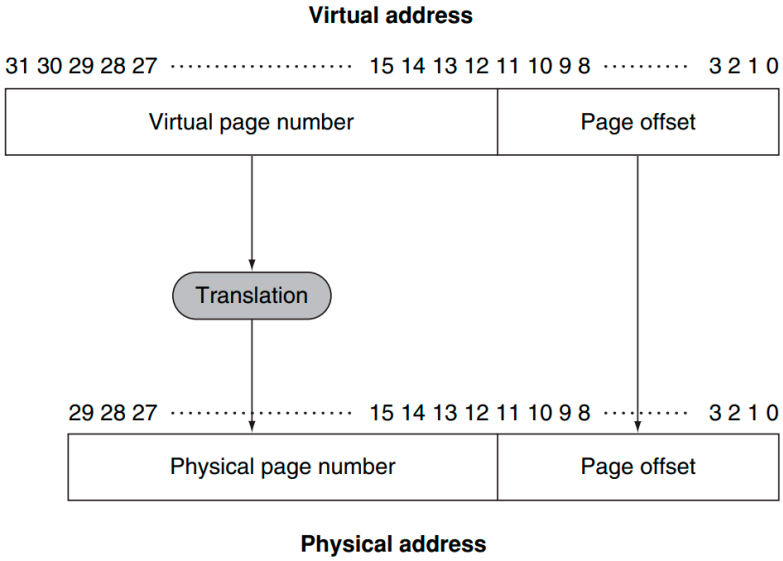
Virtual Memory

The main memory is usually acting as a “cache” for the secondary storage, this technique is called **virtual memory**. For example, if we are using *virtual machines*, the memory which is used by the machines must be available all the time, so then all the active blocks used by them (virtual machines) are stored in the main memory. The behavior is similar as it was with cache and main memory.

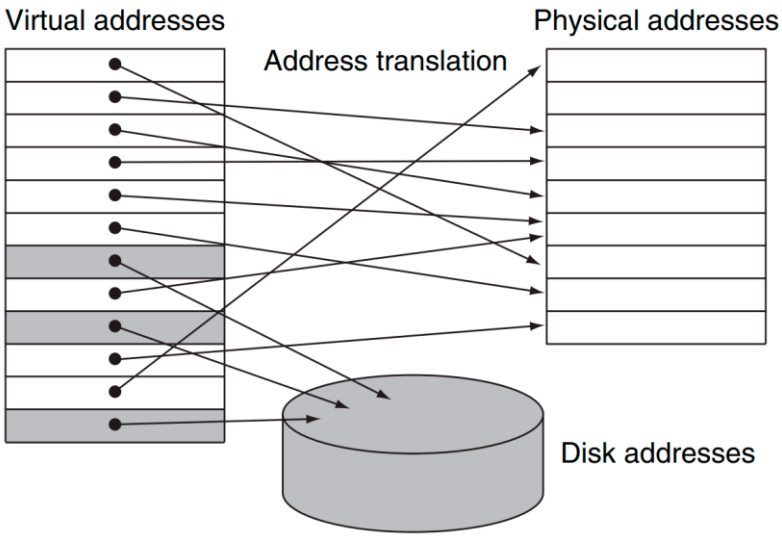
Virtual memory is implementing the translation of a program’s address space to **physical addresses**, this translation process enforces **protection** of a program’s address space from other virtual machines (in case we have several machines in one computer or another program which may access that address). When we refer to “physical” address, we refer to the main memory accessed by the processor.

Virtual memory is also allowing programmers to have programs bigger than the size of the main memory.

Although the virtual memory and caches share similar concepts, there are still some differences due to terminology. A virtual memory block is called **page**, a virtual memory miss is called **page fault**, with virtual memory the processor produces **virtual addresses** which are translated by a combination of hardware and software to *physical addresses* that can be used by main memory. This process is called **address mapping** or **address translation**.



Loading programs for executing is simplified by the virtual memory with **relocation**. This relocation allows us to load the program anywhere in the main memory, but the address in virtual memory is broken into 2, **virtual page number** and **page offset**.



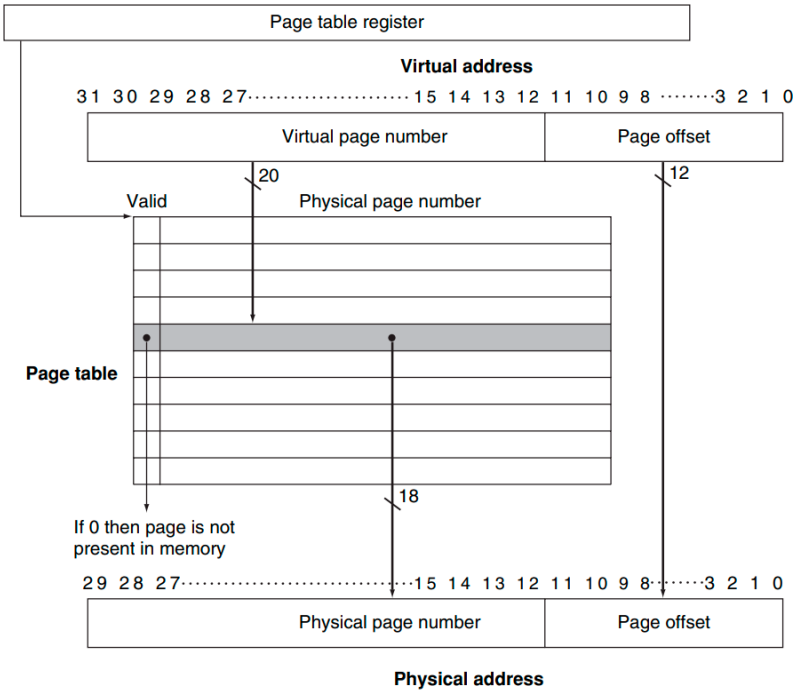
In this schema we can see the translation from a *virtual page number* to a *physical page number*. The number of bits in the page offset field determines the page size, also the number of addressable pages with the virtual machine can be higher than the number of addressable pages with the physical address.

The main goal of virtual memory is to prevent *page faults* because the penalty of one *fault* is enormous. There are several decisions regarding the design of virtual memory due to this prevention:

- Page sizes should be large enough, so the *miss rate* will be less.
- Software can be used to apply clever algorithms to place pages to reduce the *miss rate*.
- Write-through is not used in virtual memory because it will take too much time. It is used write-back.

We can also use **fully associative** placing for the *pages*. If we allow this, the operating system can then choose to replace any page it wants when a page fault occurs. But searching will be quite hard, that is why the way we locate pages is by using a table that indexes the memory; this structure is called **page table**.

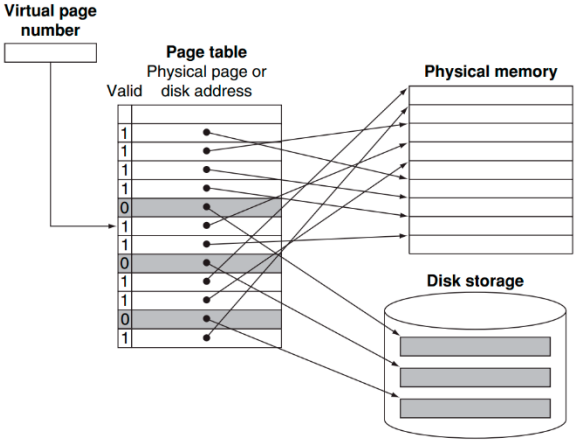
A page table is indexed with a page number from the virtual address to discover the corresponding physical page number. To indicate the location of the page table in memory computers use a register that points to the start of the page table; we call this the **page table register**.



Page faults occur when the valid bit for a virtual page is off. When this occurs, the operating system gets control, and it must find the page in the next level of the hierarchy and decide where to place the requested page in memory.

If a *page* is requested the operating system before must be keeping track of the location on this of each page, also the operating system doesn't know the moment when a page will be replaced, for this reason the operating system usually creates space on flash memory or disk for all the pages of a process. This space is called the **swap space**. When the process is created, it is also created a data structure containing all the virtual pages stored in disk. This structure may be part of the page table.

For the replacing policy, operating systems use **LRU** (*last recently used*). When a replacement is needed, the operating system writes in the *swap space*.



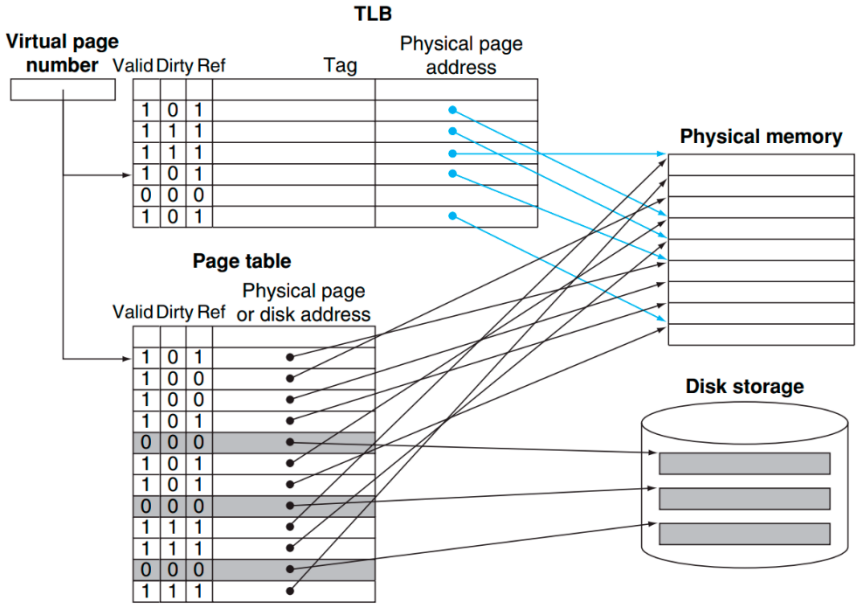
In this schema we can see how the page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level of hierarchy.

We can see how the operating system is keeping track of the location of each page.

The virtual page number is used to index the page table.

We can also see the valid bit, if it is off, it means that the page is at the specified address in the disk, if it is on, it means that the page table is giving the physical page number corresponding to the virtual page.

How do we translate the addresses? Do we have to do it every time? The answer is **no**. Instead of translating every time every address, modern computers have a buffer which is called **translation-lookaside buffer (TLB)** or **translation cache**.



As we can see the **TLB**, has stored the corresponding physical address of a virtual address. So, every time that the processor is going to ask for a translation (virtual-to-physical), the processor is going to check first the TLB.

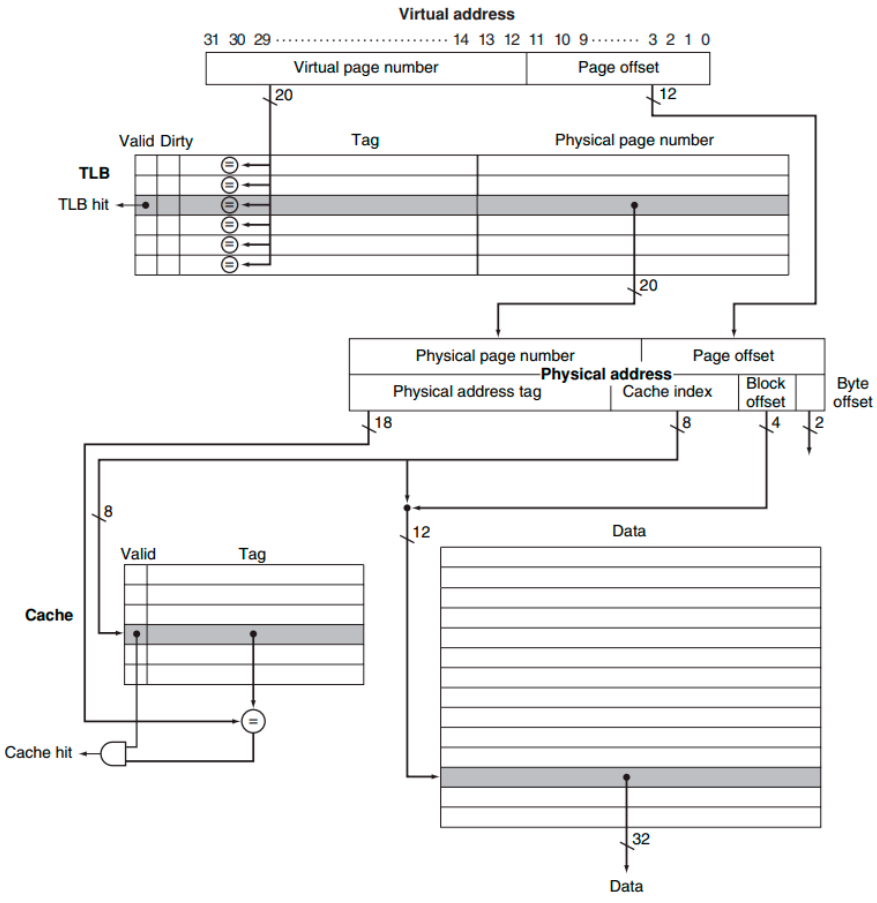
We can also see that the TLB has additional fields such as, **dirty** or **Ref (reference bits)**.

So if we are looking for a page the processor will go to the TLB, if it gets a hit, that physical address stored in the TLB is going to be used and the reference bit will be turned on. If the processors is performing a write the dirty bit will be turned on as well.

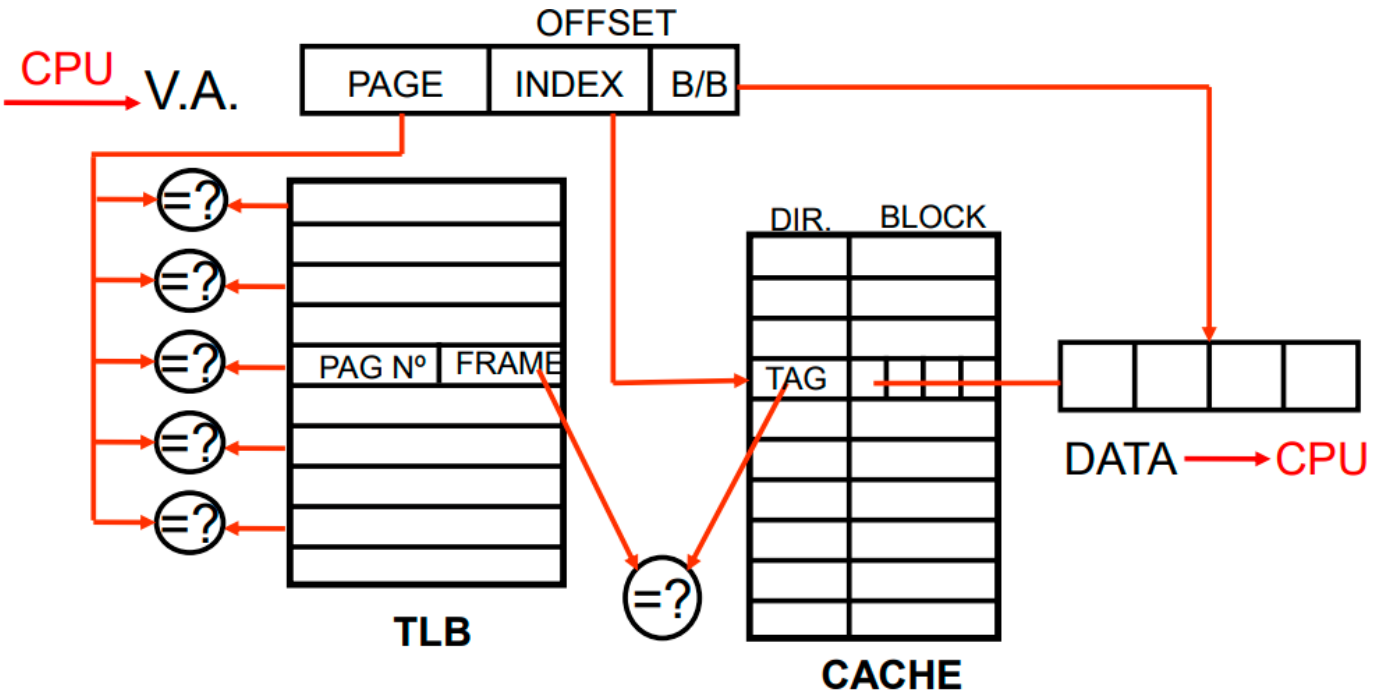
If we get a miss, the processor will have to know whether is a *page fault* in memory or just a *TLB miss* in the TLB (the page is in memory) , if it is a *TLB miss* the processor will try to load the translation in the TLB and then reference the TLB again, if it is a *page fault*, the operating system is going to generate an exception.

For TLB's there are plenty of designs, small, large and some ones which we are familiar with like *fully associative*. *Fully associative* has a very mall miss rate and TLB are usually small so fully associative mapping cost is not too high.

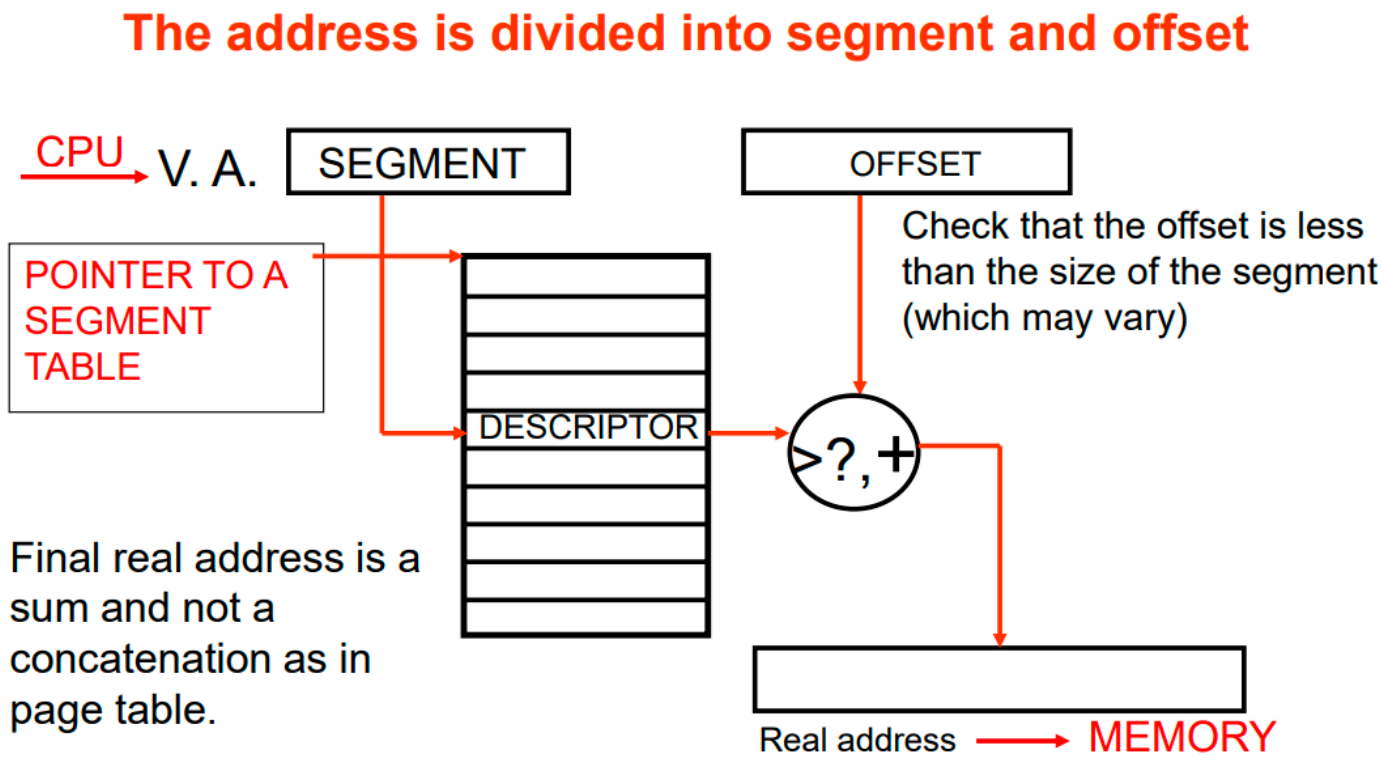
How do cache and TLB work together? For this we still have the *hierarchy*, the virtual address is translated by the TLB and sent to the cache where the appropriate data is found and finally set back to the processor.



There is another way where the TLB and the cache work together. This by **parallel access**. The TAG of the cache is compared with the FRAME of the TLB.



Another way of translating is by using a **segment table**.



The **descriptor** contains:

- **Segment start address**: It is added to the offset to compute the real address.
- **Segment size**: It must be greater than the offset.
- **Bits to control**:
  - **Present bit in main memory**
  - **Protection bit**: It protects the page against write operations (it usually refers to the code segment).
  - **Exclusion bit**: Does not allow the access of another process, this grants system security.
- **Bits for replacement algorithms**: Depending in the replacing technique used we will have different data.

There is also a protection mechanism that must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process intentionally or unintentionally. For that there is a **write access bit** in the TLB that can protect a bit for being written.

If the cache is accessed with a virtual address and pages are shared between processes, there is the possibility of **aliasing**. Aliasing occurs when there are two virtual addresses for the same page. This lets another process write in the same address but accessing another virtual address.