Two aspects for determine **Memory Speed:**

**·Latency:** The *delay* from starting an operation to receiving the result.

**·Bandwidth:** The *rate* at which data can be moved. bits/second

Address Space**:**

**·**Memory lives in an address space. With an **N-bit** address, there are **2N** addressable locations. a 32-bit architecture can address 232 = 4,294,967,296 locations .

Data Bus**:**

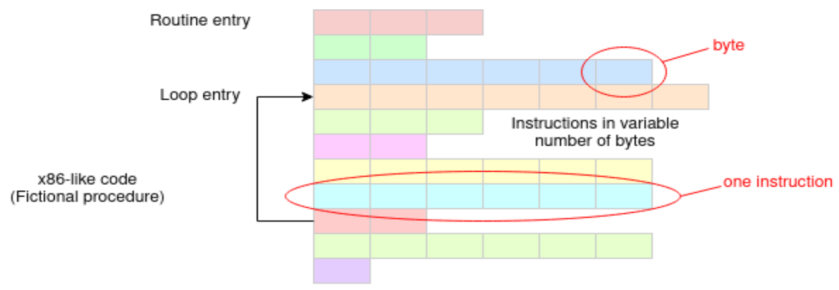
Most numeric variables need more than the 28 = 256 values a byte provides. Thus they comprise several bytes. Being sensible, these bytes will be stored **at adjacent addresses**. Moving these one-at-a-time would be slow so it is usual to have a *wider* **data bus** to move the bytes ***in parallel***.

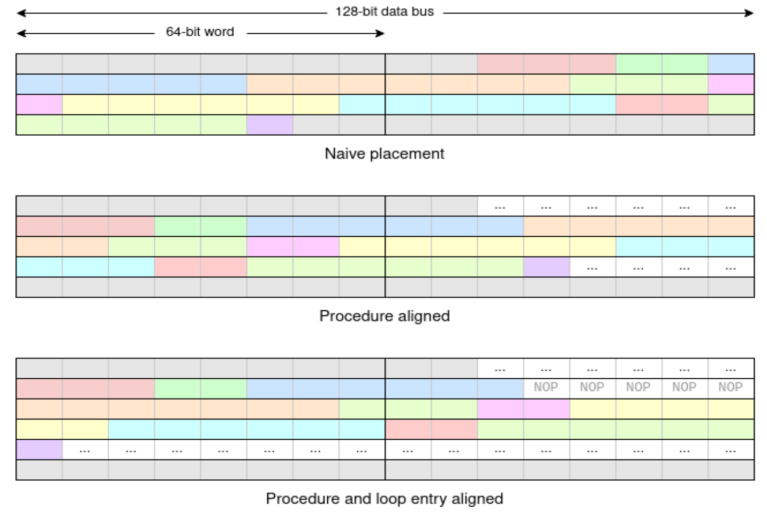
Alignment**:**

**·cache line**: Place data structures (dynamically or at compile time), “malloc” can be aligned on more coarsely aligned boundaries.

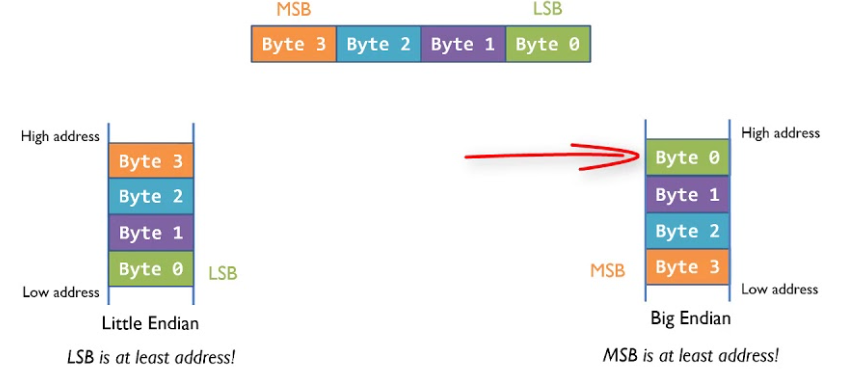
**·entry points** (i.e. branch targets) allows more object code bytes/words to be fetched if the bus width(link) permits it.

**·**The problem is even more acute with **instruction sets** - the **x86** is a good example - which have variable length instructions. Starting a section of code at an aligned address means the first fetch will capture the most useful bytes. Sometimes codes, particularly loop entry points are padded out for speed. The saving over the execution time of a particular loop will usually more than make up for the NOPs - especially as NOPs with different length op. codes can be used to reduce the instruction count.





Little Endian vs Big Endian**:**



Control signals in Bus**:**

**·**an **indication** of the direction of a given transaction (write/store or read/load), or an **indication** if the bus is active or not.

**·**for write operations of different sizes, there needs to be some **indication** of the transfer size so that the memory system does NOT modify bytes it should NOT.

Bus Implementation**:**

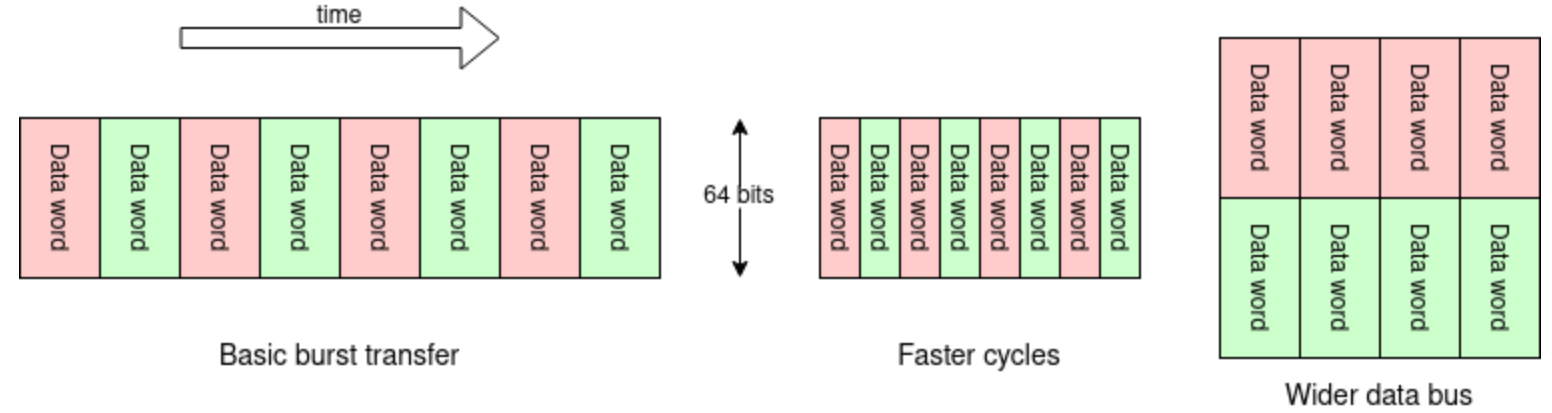
**·Bandwidth:** The *rate* at which data can be moved. bits/second

**·Latency:** The *delay* from starting an operation to receiving the result.

**·Read Latency:** the time from wanting some data (from a determined address) and receiving that data.

**·**caching can be used to reduce read latency ; write buffering can be used to reduce write latency.

**·**Bandwidth can be reduced by cycling the bus faster(not useful) or making the bus wider(useful when it is known that *coherent blocks* of data such as cache lines are being moved; it does NOT help with individual ‘random’ transfers).



I/O spaces**:**

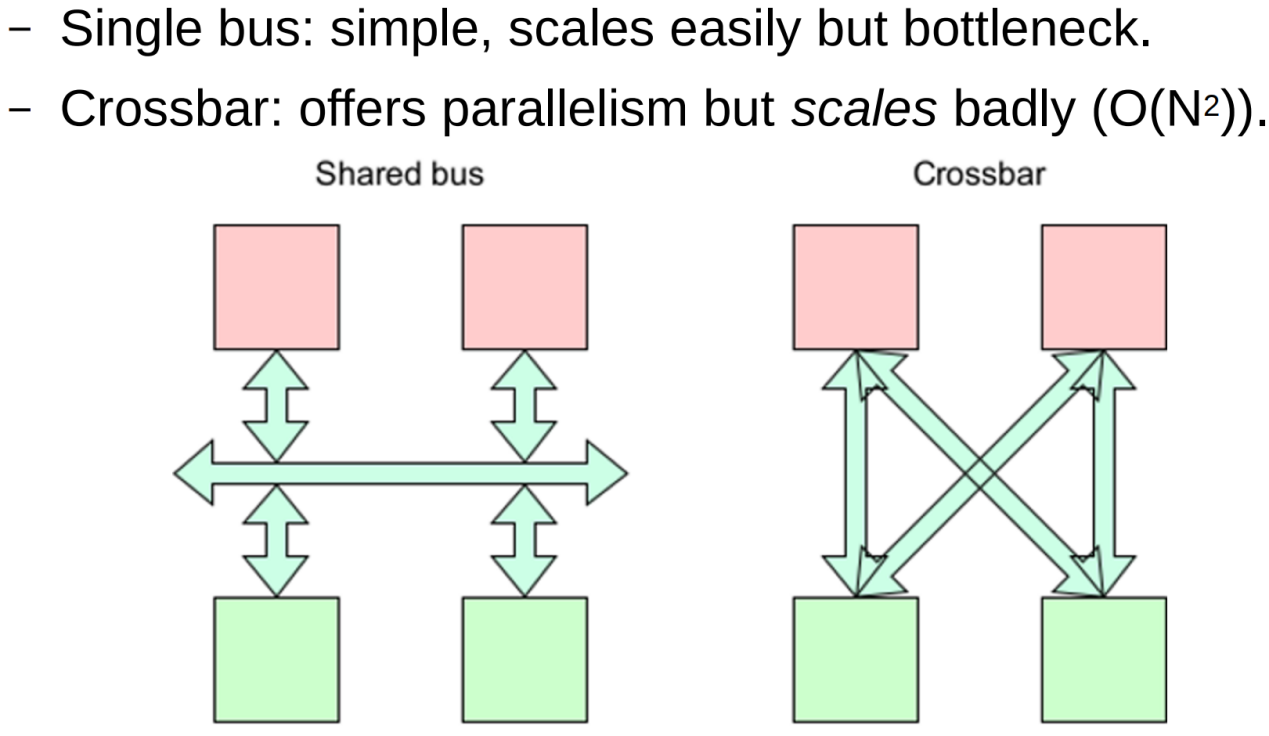
**·I/O spaces** are mediated by ‘**peripherals**’ - specialized devices which translate software operations into the appropriate sequences of signals.

**·peripherals** will have a number of specialized registers which are used to monitor and control external quantities. For example, if controlling a motor there could be a register which is written to to set the desired speed, and another where the actual speed could be monitored. These registers are memory mapped.(i.e. they have unique addresses in overall address space).

**·Input bit** changes autonomously. Ensure areas of address space containing peripherals are NOT cached.

Bus Signals**:**

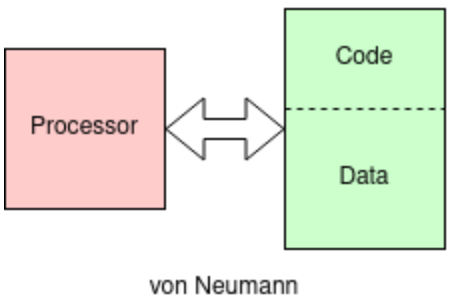
|  |  |
| --- | --- |
| **wait** | Typically there is some timing signal to slow operations when the memory cannot respond immediately. |
| **fault/abort** | Something went wrong. This is used primarily from an MMU for virtual memory *page faults* but may also signal I/O problems etc. |
| **locking information** | For multiprocessor operations. |
| **sequence information** | Hints about the relationship of successive addresses can help accelerate some memory accesses. |
| **privilege state** | An indication of the processor *mode* - i.e. is it currently running an application, O.S. etc. Access to some addresses are only allowed in particular modes. |



von Neumann Architecture**:**

**·**a unified memory holds both data & instructions.

**·**more convenient for programmers: the available memory can be partitioned dynamically into code and data segments as is currently appropriate. Data and code can be interchanged, e.g. loading then running a binary file or compiling a programme.



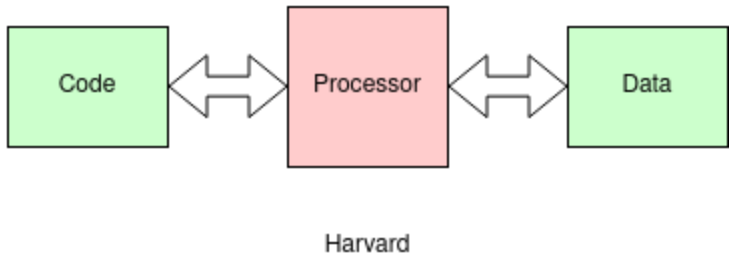
Harvard Architecture**:**

**·**data & instructions are in separate address spaces.

**·**more engineering flexibility: data and instruction words can be different sizes. In particular it allows instruction fetches in parallel with data movement, helping to overcome the memory bottleneck.

**·**higher memory bandwidth through parallel access

**·**allow memories of different widths.



Memory Hierarchy**:**

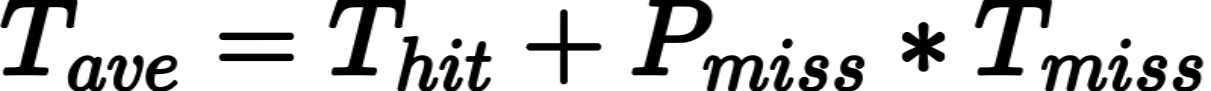
**·**Small Memories are fast; Big Memories are slow.

**·**To maximize performance, we want the frequently used locations to be fast.

Locality**:**

|  |  |
| --- | --- |
| **Temporal Locality** | if an address is used, then maybe it will be used again. |
| **Spacial Locality** | if an address is used, then maybe its nearby addresses be used. |

Hit & Miss**:**



**Average cycle time = 1 + miss rate × miss penalty**

wps: average memory access time

C:/Users/HELLO/AppData/Local/Temp/wps.toTktMwps: hit time

C:/Users/HELLO/AppData/Local/Temp/wps.WFsBWYwps: miss rate (probability of a cache miss)

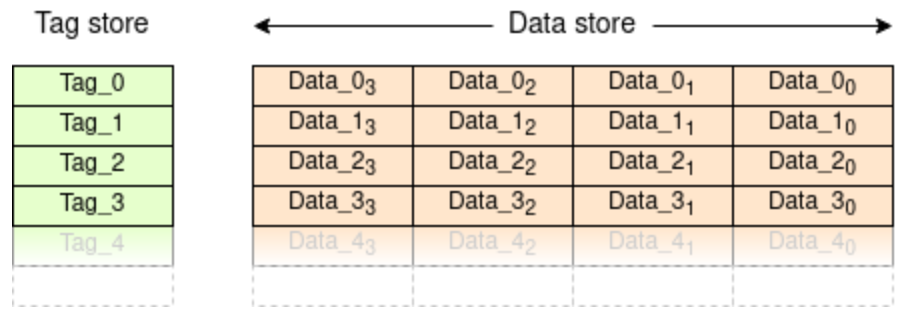
C:/Users/HELLO/AppData/Local/Temp/wps.JwbNvAwps: miss penalty

**·**a **cache miss** occurs when a program access data which is NOT stored in the cache, then must access data from the slower main memory. opposite to ‘cache hit’

**·cache penalty** is the time taken to retrieve data from the main memory when a cache miss occurs

**·hit time** is the time it takes to access data from the cache when a cache hit occurs. When a program accesses data, the processor first checks the cache to see if the data is stored there. If the data is found in the cache, it is called a cache hit, and the processor can access the data quickly.

Cache Architecture**:**

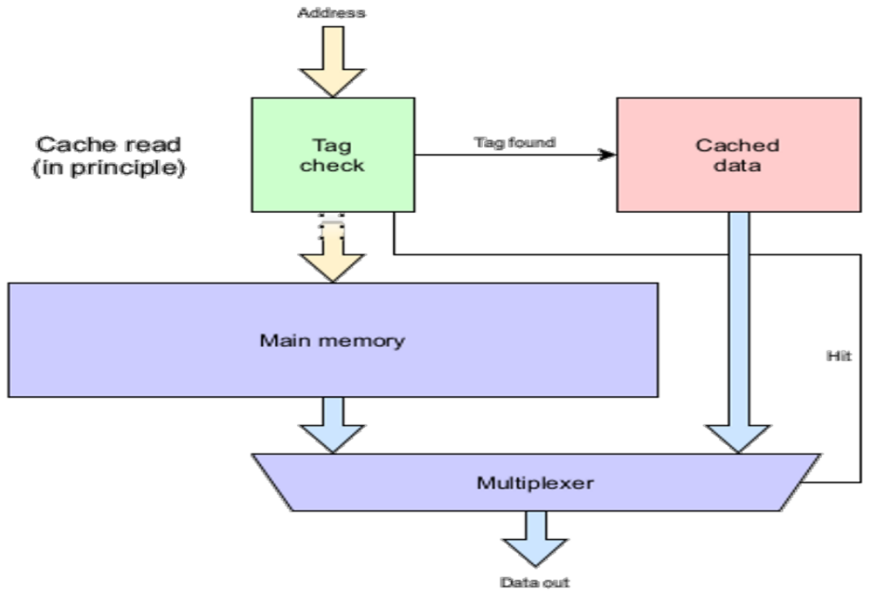


**·data memory** holds a copy of a subset of the larger storage space.

**·tag** identifies which elements are currently stored; a corresponding **data field** which holds what is held in the tagged locations (i.e. data).

**·**Many caches exploit spatial locality by sharing a single tag across a block of adjacent addresses; this saves storing lots of tags when they would typically have repeated values. This block is known as a **cache line**.

Cache Operation**:**



**1.** Address is checked in tags.

**2.** If address recognized, corresponding data read; 'Hit' sends data to output.

**3.** If address not recognized, address sent on to memory.

**4.** Memory read and output if needed.

Fully Associative Cache**:**

It is flexible, because any data element can be cached in any cache locations. It gives the best hit rate. However, input address needs to be compared with all the tags on every access, which is expensive, especially in energy. It is slow because reading data has to wait until the correct tag is determined.

Direct Mapped Cache**:**

It saves energy, because it makes a cache look-up easier since the address being checked only needs to be compared with one entry whose position is known from the incoming address. However, it is NOT flexible, because one data element can only be cached in one particular cache locations. It decreases the hit rate. It is fast, because the data read is typically done speculatively, in parallel with the tag comparison in the hope that it will be a hit. In many circumstances a hit is confirmed and the data is already ready to return.

Set Associative Cache**:**

A two-way set associative cache is basically two direct mapped caches in parallel, thus offering two possible places any datum may be cached and preventing some conflicts. The sets will each be half the size.

Cache Misses**:**

|  |  |
| --- | --- |
| Compulsory Misses | a block of data is accessed for the first time. unavoidable. |
| Capacity Misses | Cache is full and more space needed. |
| Conflict Misses | when two or more blocks of data map to the same cache line or set.  NOT in **fully associate caches**  Often in **direct mapped caches** |

*Every miss imposes a performance penalty.*

– fully associative cache: slower but miss less often.

– direct mapped cache: faster but miss more often.

more about Cache**:**

**·**When a cache miss occurs, a check is made that the requested address can be cached. I/O peripheral locations should NOT be considered for caching.

**·**In a **direct mapped cache**: there is only one place the fetched line can go. In a **fully associative cache**: any line can be chosen. In a **set associative cache**: the set the line is going into must be chosen; within the set the position is still predetermined by the address.

Allocation Algorithm**:**

**·Random Replacement:** pick a random line.

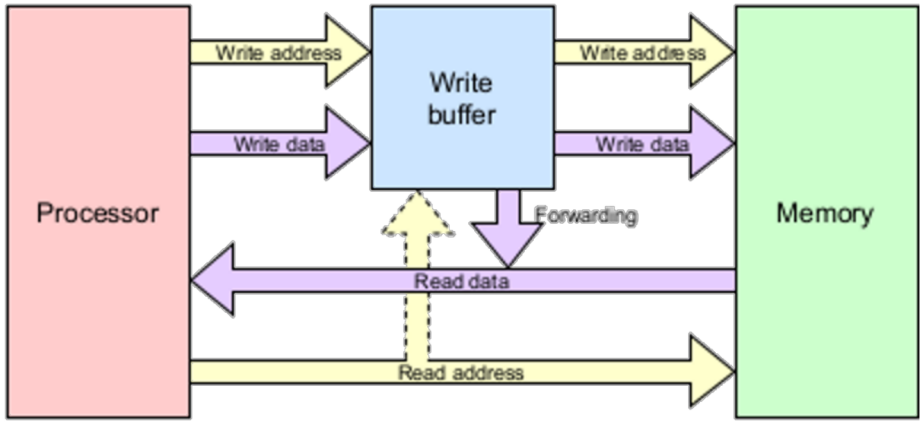
**·Cyclic Replacement:** each block of memory is mapped to a fixed location in the cache. When the cache is full and a new block needs to be loaded, the cache replaces the oldest block, using a round-robin approach. *Disadvantage:* frequently accessed blocks may be replaced by less frequently accessed blocks, leading to poor cache performance.

**·LRU(least recently used) Replacement:** each block is assigned a “usefulness” score based on how recently it has been accessed. When the cache is full and a new block needs to be loaded, the block with the lowest usefulness score (i.e., the block that has been unused for the longest period of time) is evicted. *Disadvantage:* require more storage; hard to implement

**·**When picking a cache line, it is good to fetch the new line, because latency can slow everything down. The fetch is a cache line so the fetch operation is a burst of data with some confidence that spacial locality will reduce the total number of misses (and resulting latency penalties).

**·**When the fetch is taking place, the processor is stalled and then lose performance. The latency is serious.

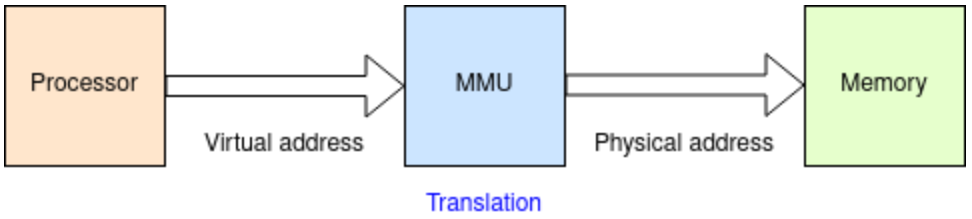
write buffer / write buffer**:**



**·**a temporary storage area to optimize write operations to memory or other storage devices. When a CPU or other device writes data to memory or a storage device, the write buffer temporarily stores the data before it is written to its final destination.

**·**Improve system performance by allowing the CPU or other device to continue executing other instructions while the write operation is being completed in the background. Without a write buffer, the CPU would have to wait for the write operation to complete before proceeding with the next instruction, which can result in reduced system efficiency and increased latency.

MMU memory management unit**:**

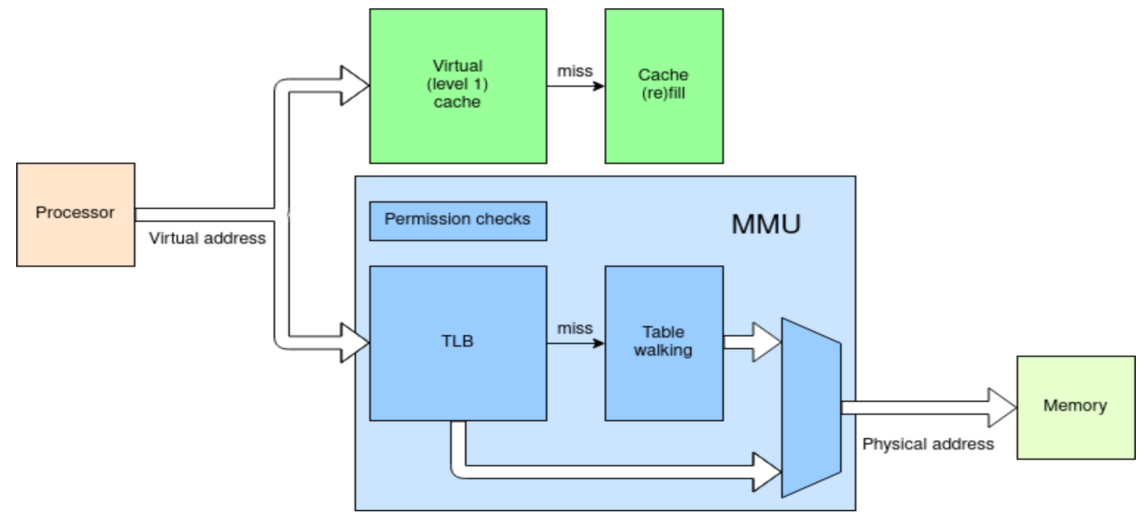


**·**Translate address & Protect memory.

**·**Translate virtual addresses into physical addresses by using a page table(stored in memory) that maps each virtual address to its corresponding physical address. However, the translation is serial, cause latency, which can be reduced by a virtually addressed cache such as TLB.

**·**Ensure that the memory is properly allocated and that processes are not able to access memory that has been allocated to other processes or the operating system.

TLB translation lookaside buffer**:**



**·**a **cache** of **address translations**.

**·**hold a subset of the possible page translations.

**·**When a program attempts to access a virtual memory address, the **TLB** is checked first to see if the virtual-to-physical address translation is already cached. If the translation is *found* in the TLB, the corresponding physical memory address is used to access the memory. If the translation is *not found* in the **TLB**, the processor is stalled, a page table lookup is performed to find the corresponding physical memory address, and the result is stored in the TLB for future use.

MPU memory protection unit**:**

**·**Provide permission checks for areas or memory, without the virtual-to-physical translation. Then it can prevent software components from accessing or modifying memory that they are not authorized to access.

Cahcability**:**

**·**MMU & MPU can mark which addresses can be cached or not.

**·**active DMA transfers and address space for I/O cannot be cached because they will change contents without processor actions

SRAM static random access memory**:**

**·**store data using transistors and flip-flops

**·**fast

**·**quick & random access

**·**used in all cache levels

**·**large power consumption

DRAM dynamic random access memory**:**

**·**store data using capacitors

**·**slow

SDRAM synchronous DRAM**:**

**·**DRAM with a state machine to make access faster and more convenient.

Flash**:**

**·**permanent store.

**·**used for long-term storage such as file-systems.