#uni/semester3/Betriebssysteme/chapter19/chapter-notes

➜ paging requires a large amount of mapping information. Because it is generally stored in physical memory, paging requires an extra memory lookup for each virtual address. Going to memory for translation information before every instruction fetch or explicit load or store makes things slow.
➜ To speed it up we need hardware support. We are going to add what is called translation-lookaside buffer or TLB. It is a part of the memory management unit (MMU), and is a hardware cache of popular virtual to physical address translations. A better name would be address translation cache. At each virtual memory reference, the hardware first checks the TLB to see if the desired translation is there. If so, the translation is performed quickly without consulting the page table which has all translations.

1. TLB Basic Algorithm
   ➜ Assumptions for algorithm
      ➜ linear page table (page table is an array)
      ➜ hardware-managed TLB (the hardware is responsible for page table accesses)
   ➜ First extract VPN from VA and check if there is a translation for it in TLB
   ➜ If yes, we have a TLB hit. Then extract the PFN from TLB entry, combine it with offset from VA, check protection bit and then access memory
   ➜ If no, we have a TLB miss. Then access page table to find the translation, check if virtual memory reference is valid and accessible, update the TLB with the translation. Retry the instruction, but this time the translation is found in TLB. This is very costly because of extra memory reference PTE = AccessMemory(PTEAddr) in line 12.
   ➜ the TLB is built with the premise that hits are the common cases. Because misses lead to more memory accesses (thus are slow), we want to avoid TLB misses as much as possible.

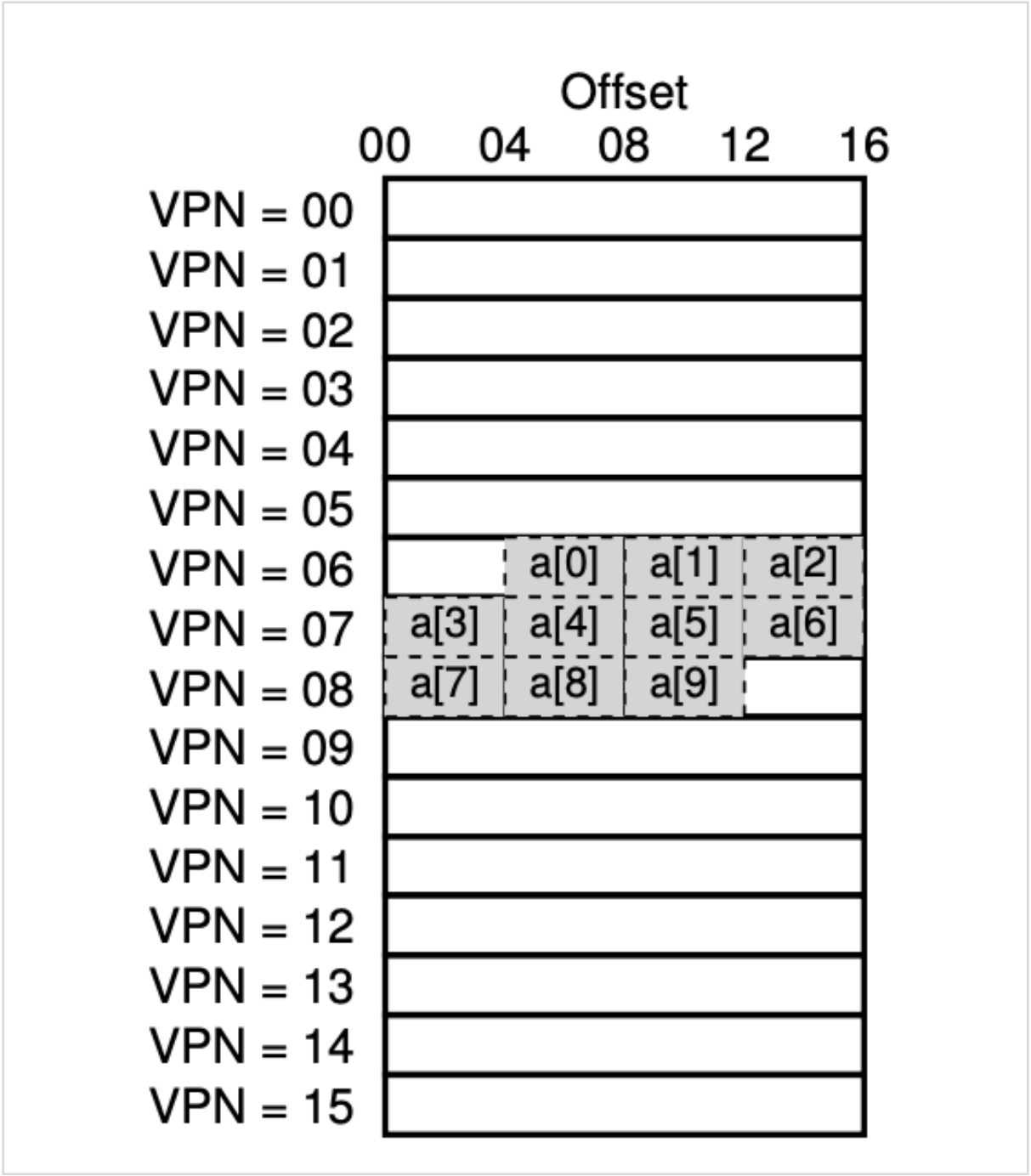
```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                   // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

2. Example: Accessing An Array
   - ➜ Array of size 10, with 4 Byte integers
   - ➜ Array starts at virtual address 100
   - ➜ VA is 8 bit, 4 bit offset and 4 bit vpn.
   - ➜ We use a loop to access the array for the first time In the program

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

   - ➜ we ignore in this case other memory accesses like sum and i.
   - ➜ a[0] means load 100 will happen, this is VPN 6. Page wasn't translated before, so there will be a TLB miss. Thus a[1] and a[2] will be a hit and the rest so on.



➜ TLB hit rate = hits / num of accesses = 7 / 10 = 70%

➜ TLB improves performance due to spatial locality. The elements of the array are tight together and thus only the first access to an element yields a TLB miss. Larger page sizes, means less TLB misses in this case.

➜ If after the loop is finished, the program accesses the array again, we would likely see an even more hits because of temporal locality. Which means the quick re-referencing in time.

➜ spatial and temporal locality are program properties. Therefore if a program respects this locality the TLB hit rate will be high.

➜ temporal locality means if data has been recently accessed it will likely be re-accessed soon in the future.

➜ spatial locality means if data is accessed at an address x, it will likely soon access memory near x.
➜ hardware caches take advantage of locality by keeping copies of memory in small, fast on chip memory. Why not use bigger caches? Because of laws of physics. Large cache is automatically slow.

3. Who Handles The TLB Miss?
➜ in the old days hardware had complex instruction sets (CISC) which handled TLB miss entirely. For this the hardware needs to know the page table base register (location) and the format of the page table. The hardware would then extract the desired translation, update the TLB and retry the instruction. X68 Architecture has hardware managed TLBs which uses fixed multi-level page table. CR3 points to current page table.
➜ Modern RISC architectures have a software-managed TLB. On TLB miss the hardware raises an exception which raises the privilege to kernel mode and jumps to a trap handler. OS looks up the translation in the page table, updates the TLB with specialised instructions and returns from trap. Then hardware retries the instruction.
➜ return-from-trap needs to resume execution at the instruction that caused the trap. Return-from-trap from system call returns to after the instruction that caused the trap. So depending how an exception or trap was caused, the hardware must save a different PC when trapping into the OS.
➜ When running the TLB miss handling code, the OS must be careful not to cause an infinite chain of TLB misses. One way would be to keep TLB miss handlers in physical memory, thus not being unmapped and not subject to address translation. Another way would be to reserve some entries in the TLB for the handler code. These wired translation always hit in the TLB.
➜ Advantage of software-managed approach is flexibility, as it can use any data structure to implement the page table. Another one is simplicity. See codes in chapter (19.3 and 19.1). The hardware doesn't do much on a miss, just raise an exception and let the OS do the rest.

4. TLB Contents: What's In There?
➜ a typical TLB might have 32, 64 or 128 entries and be what is called fully associative. It means that the translation can be anywhere in the TLB and that the hardware will search the entire TLB. An entry might look like this: VPN | PFN | other bits
➜ looks in parallel means Virtual Memory: 13 TLBs and Caches - YouTube
➜ TLB is a full associative cache
➜ TLB has a valid bit, which says if an entry has a valid translation
➜ protection bits, how a page can be accessed. Code might be read and execute, while heap read and write.

➜ Other bits like address space identifier, dirty bit and so on.

➜ valid bits in TLB and those in page table are not identical. When a PTE is marked invalid it means that the page has not been allocated by the process. If illegal access, then traps to the OS/

➜ A TLB valid bit, shows if a TLB entry has a valid translation. At boot time all entries in TLB are set to invalid, because there are no translations cached yet. TLB valid bit helps with context switch. By setting all TLB entries to invalid, the system ensures that the new process does not accidentally use a translation from a previous process.

5. TLB Issue: Context Switches

➜ hardware or OS (or both) must ensure that one process does not use translations from some previously run process.

➜ Example if the 10th virtual pages of two processes are in the TLB but translate to different PFN

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

➜ Problem is that hardware can't distinguish which entry is meant for which process.

➜ To make the TLB support virtualisation across multiple processes there are some possible solutions. One would be to flush the TLB on context switch, by empyting it before running next process.

➜ On a software based system this could be done with an explicit and privileged hardware instruction.

➜ On a hardware managed TLB, this flush happens when page table base register is changed. In both cases the flush sets all valid bits to 0.

➜ Problem with this solution, is that when process runs there will be a lot of TLB misses which have a high cost.

➜ To reduce this overhead, hardware support is added to enable sharing of the TLB across context switches, by adding an address space identifier (ASID) to the TLB.

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

➜ To differentiate the hardware also needs to know which process is currently running.

➜ Two pages of different processes which map to the same PFN:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

➜ two processes share a page 101 (a code page e. g.)

➜ Sharing code pages (binaries or shared libraries) reduces the number of physical pages in use and thus reducing memory overheads.

6. Issue: Replacement Policy

   ➜ when adding a new entry to TLB which one should we consider to replace? (Cache replacement). Goal is to minimise the miss rate and increase the hit rate.

   ➜ One approach is LRU, least recently used, which uses the locality to assume that an entry that has not been recently used is a good candidate for eviction.

   ➜ Another one is a random policy, which evicts an entry at random. It is simple and avoids some unreasonable behaviours. For example when looping through n + 1 pages, with a TLB size of n, the LRU would produce TLB misses on every access, where random does better.

7. A Real TLB Entry

➜ modern system that uses software managed TLB. Here is an entry:

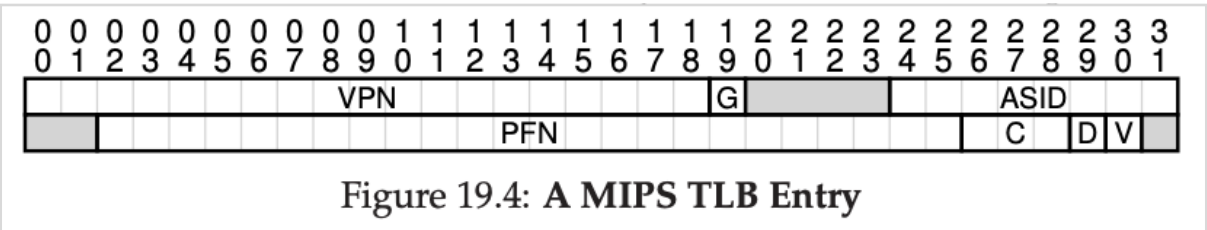


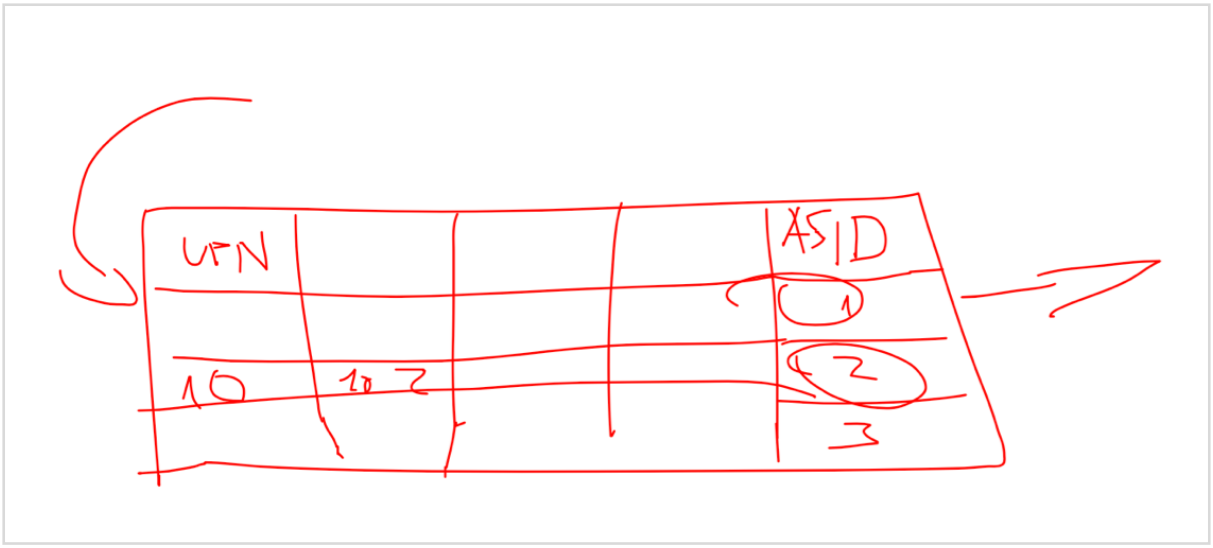
Figure 19.4: **A MIPS TLB Entry**

➜ 32 bit virtual addresses with 4 kb pages. This means 20 bit for VPN and 12 for Offset. In reality we only get 19 bits for VPN, because user addresses come only from the half the address, the rest being reserved by the kernel.

➜ 24 bit for PFN

➜ G bit is a global bit, used for pages that are globally shared among processes. If it is set, the ASID is ignored.

➜ 8 bit ASID to help OS distinguish between address spaces. When it runs out, I think OS should the unused space to enable more. Or maybe it doesn't help, because the case is too small. Or maybe it starts to replace some, with some kind of policy. There is ASID map, which maps the ASIDs that are used. It starts to reassign the ASIDs.



➜ 3 coherence bits to determine how a page is cached.

➜ dirty bit, when is set when a page has been written to.

➜ valid bit, it tells the hardware if there is a valid translation in the entry.

➜ also a not shown page mask field, which supports multiple page sizes

➜ some TLB entries are reserved by the OS. A wired register can be set by the OS to tell the hardware how many entries in TLB to reserve for OS. OS uses these for access during critical times where a TLB miss would be problematic (TLB miss handler)

➜ TLB has privileged instructions that update the TLB

- ➜ TLBP, checks if a particular translation is in the TLB
- ➜ TLBR, reads content of entry into registers
- ➜ TLBWI, replace a specific entry
- ➜ TLBWR, replace random TLB entry

➜ random access memory implies that you can access any part of RAM just as quickly as another. If a page is not mapped by TLB, then it will be much slower. So RAM isn't always RAM. This is called Culler's Law.

8. Summary

➜ exceeding the TLB coverage means to exceed the number of pages the fit into the TLB. This leads to slowness.

➜ One solution is to support larger page sizes by mapping key data structures into regions of the programs address space. Support for large pages is often exploited by data base management system (DBMS), which have data structures that are both large and randomly accessed.

➜ Physically indexed cache can be a bottleneck to the CPU, because address translation has to happen before the cache is accessed, which can slow things down.

➜ Virtually indexed cache access caches with virtual addresses which solves some performance problems, but introduces new ones.

https://www.youtube.com/watch?v=Z4kSOv49GNc&app=desktop