Operating System (OS)
CS232

Memory Management: Translation Lookaside Buffer (TLB)

Dr. Muhammad Mobeen Movania

Outlines

- Recap
- Improving paging through TLB
- TLB Basic Algorithm
- TLB example
- Why TLBs improve performance?
- Who handles TLB misses?
- TLB contents
- TLB issues
- Summary

Recap

- We saw that paging solved the external fragmentation problem
- Paging is very slow
  - Has to access page table in RAM
  - This takes place at each memory access
  - Page tables can grow very large

Speeding up paging

- Accomplished via Hardware Support
- TLB works as a cache for frequently performed virtual address to physical address translations
- TLB is a part of MMU
- Upon accessing a virtual address, before accessing a page table:
  - The hardware first tries to find a VPN in the TLB
    - If successful, it uses the corresponding PFN (thus saving the access to page table)
    - If unsuccessful, it goes and accesses the page table in RAM to get the PFN and then Updates the TLB

TLB Basic Algorithm

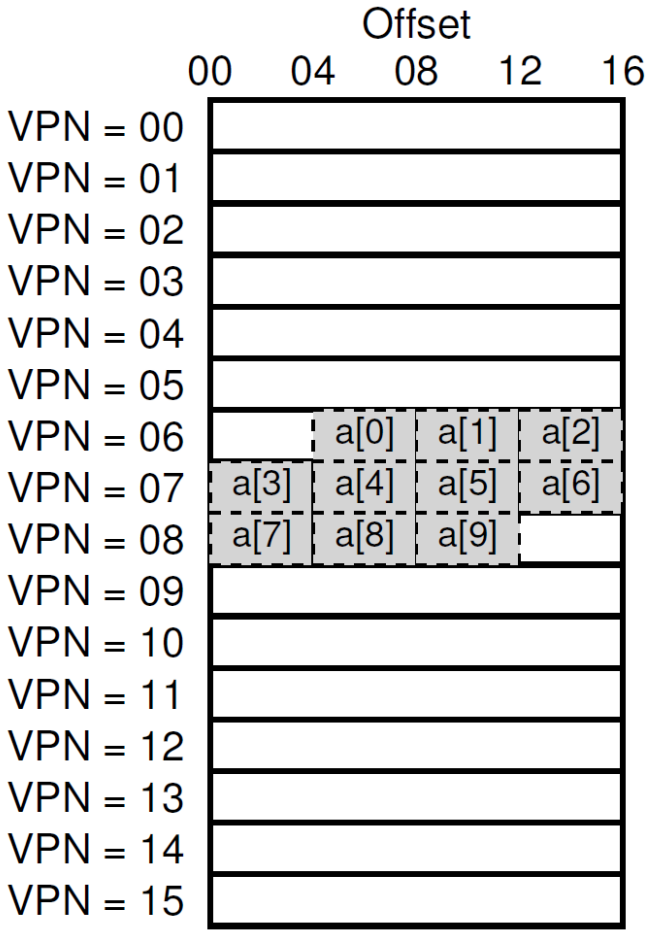
```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = 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**

TLB – example

- 8 bit address space
- 16 virtual pages 16 bytes per page
- Array of 10 ints (4 bytes each)

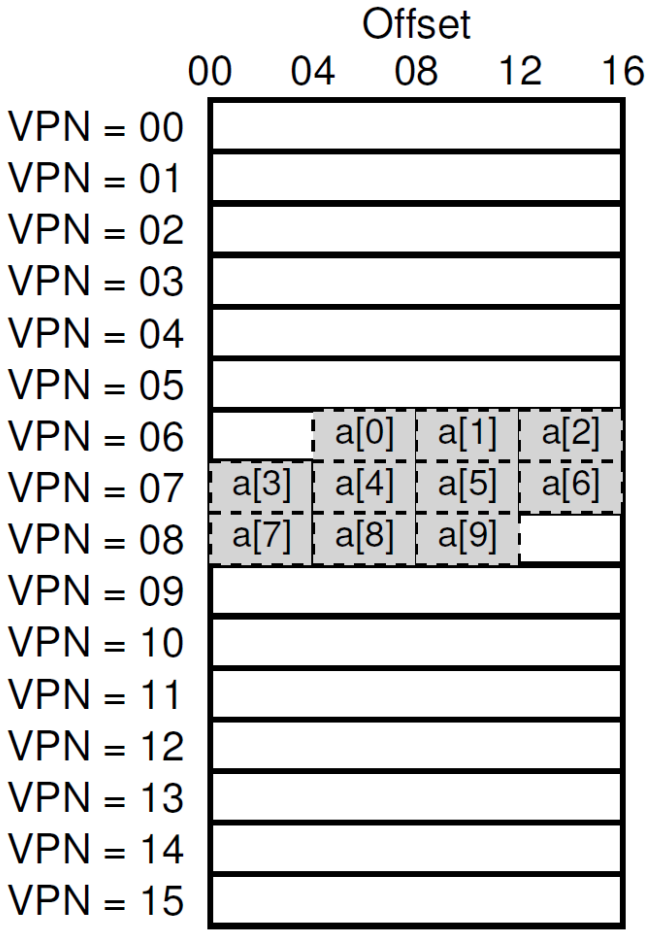
```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```



TLB – example … contd.

Initially assume that the TLB is empty.

- Access a[0] : VA=100 (VPN 6) TLB Miss
- Access a[1] : VA=104 (VPN 6) TLB Hit
- Access a[2] : VA=108 (VPN 6) TLB Hit
- Access a[3] : VA=112 (VPN 7) TLB Miss
- Access a[4] : VA=116 (VPN 7) TLB Hit
- ….
- We get
  - M H H M H H H M H H
- TLB hit rate
  - total hits/total access = 7/10 (70%)



Why TLB improves performance?

- Due to ***spatial locality*** as array elements are packed tightly into pages (i.e., they are close to one another in **space**)
- Only the first access to an element on a page yields a TLB miss
- If page size is increased, we get even fewer misses
- If the program accesses the array again after loop, with a big enough TLB we might benefit with 100% hit rate due to ***temporal locality***

Who handles a TLB miss?

- CISC
  - Usually hardware
    - Reads PFN, updates TLB, and retries the instruction
- RISC
  - Software (OS)
    - On a TLB miss, the hardware raises exception
    - The OS exception handler reads PFN from page table in memory
    - Updates the TLB
  - return_from_trap (exception)
    - This return_from_trap is different from the return_from_trap of a system call as it returns to the instruction which raised exception, executing the same instruction again.
  - What if the instruction that accesses a page table for PFN results in TLB miss?

TLB contents

- Typically 32, 64, or 128 entries
- Fully associative
  - Search can be performed anywhere in TLB in parallel
- Other bits:
  - Valid bit
  - Protection bits
  - Dirty bit
  - Address Space ID (ASID)

VPN | PFN | other bits

TLB issue – context switches

- P1 VPN 10 mapped to PFN 100
- P2 VPN 10 mapped to PFN 170

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

- Solution:
  - 1. Flush the TLB (set all valid bits to 0)
    - Costly. Misses each time context is switched
  - 2. ASID
    - Needs hardware support. ASID of current process has to be updated on context switch

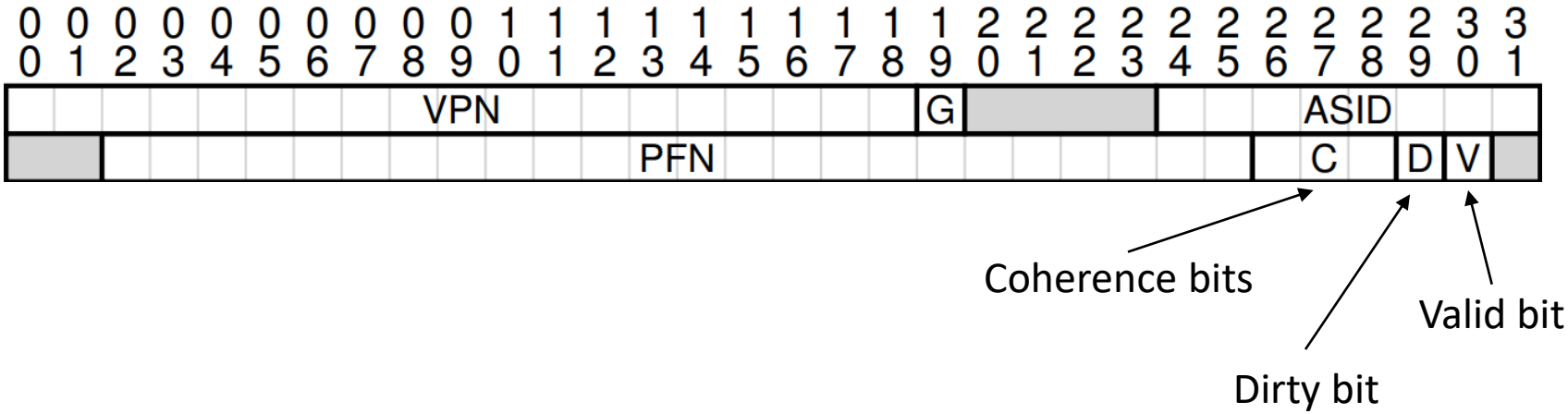
VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

TLB issue: replacement policy

- Associative cache
- When all entries are full and a new one arrives
- Which one to replace?
  - LRU
    - Corner case:
      - loop over n+1 pages in a TLB of size n, each loop we have a miss
      - Example: We have an array of 100 elements, TLB with 36 bytes per page (i.e. it can hold 9 ints), each loop accesses 10 array elements
  - Random
    - Simple to implement

TLB entry: example

- MIPS R4000
- 32 bit address space
- 4KB pages
- Global bit (G) for globally shared pages. ASID ignored.



Summary

- We saw how hardware helps make address translation faster
- TLBs act as address-translation cache
  - Allows most memory references without accessing page table in main memory
- TLBs perform poorly if
  - Number of pages a program accesses in a short period of time exceeds the number of pages that fit into TLB (programs exceeding TLB coverage)
  - Physically indexed cache is used: address translation takes place before cache is accessed which slows things down quite a bit.