OSTEP Chapter 19

ECE 3600, Fall 2022

Table of Contents

1. TLB

TLB = translation-lookaside buffer = address-translation cache (may be at least partially managed by OS software)

[vs. lower-level data caches (L1, L2, L3) handled completely by hardware]

Assuming a linear page table (i.e. the page table is an array) and a hardware-managed TLB:

```
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. Cache Example

Consider an array of 10 4-byte integers in memory, starting at virtual address 100 = 0b01100100

An 8-bit virtual address space, with 16-byte pages: VA (8) = VPN (4) Offset (4) = 0b0110 0b0100 = 6 4

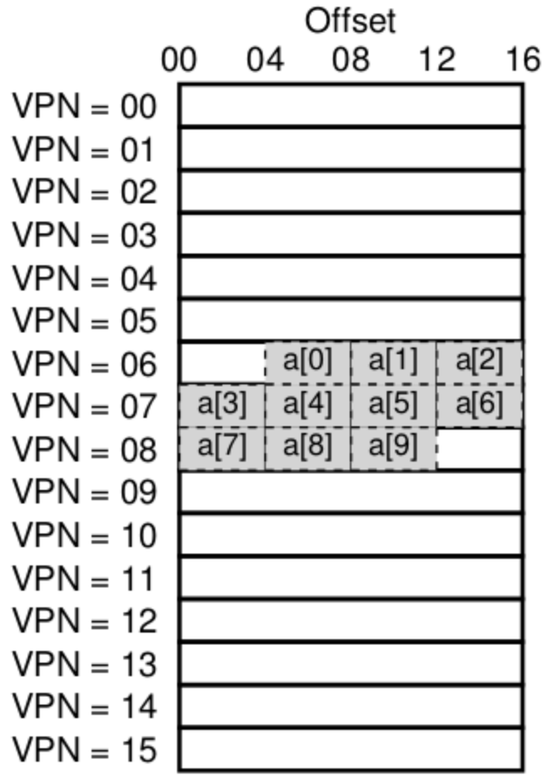

Figure 19.2: **Example: An Array In A Tiny Address Space**

(Above is data cache, TLB cache not shown)

3. Memory Hierarchy Example with TLB and L2 Cache

Virtually Indexed, Physically Tagged

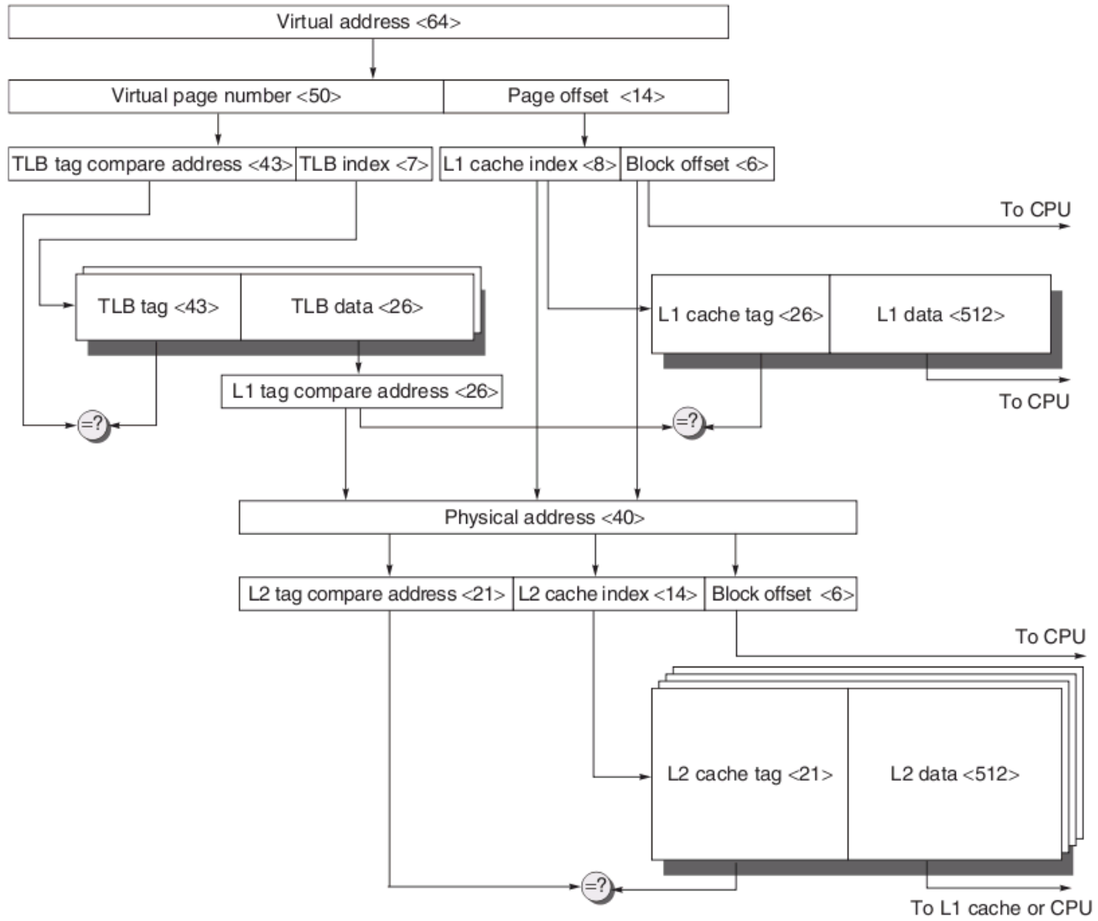

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KB, and the L2 cache is a four-way set associative with a total of 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

Note: L2 cache tag should be 20 bits (from Hennessy & Patterson)

4. OS Handling TLB Miss

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
    RaiseException(TLB_MISS)
```

Figure 19.3: **TLB Control Flow Algorithm (OS Handled)**

5. TLB Contents

TLB entry = VPN, PFN, other bits (valid, protection, dirty, address space ID, ...)

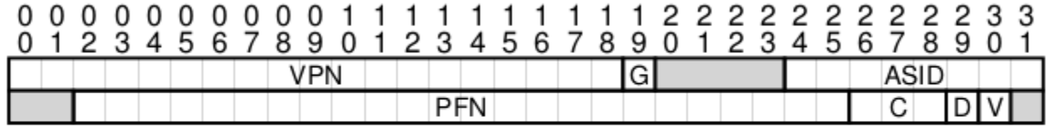

Figure 19.4: **A MIPS TLB Entry**

32-bit address space with 4KB pages.

19-bit VPN (+1 bit reserved for kernel), 12-bit offset, translated to 24-bit PFN

2^{24}*4KB = 2^{36} = 64 GB physical memory

global bit (G), 8-bit ASID, 3 coherence (C) bits, dirty bit, valid bit.

6. Measuring Cache Effects

Homework P19

Consider a 2D array with each row occupying one page of memory:

```
#include <unistd.h>
#include <stdlib.h>
...
int main( void)
{
  int nrows = 300, PAGESIZE = sysconf(_SC_PAGESIZE), ncols = PAGESIZE/sizeof(int);

  int *a = calloc( nrows*ncols, sizeof(int));
```

Note that a dynamically allocated 2D array is stored as a 1D array, so `a[row][col]` is really `a[row*ncols+col]`

Accessing the array by rows should be much faster than access by columns, due to TLB and data caching:

```
// by rows
//
 for( int row = 0; row < nrows; ++row)
   for( int col = 0; col < ncols; ++col)
     a[row*ncols+col] += 1;
```

or:

```
// by cols
//
 for( int col = 0; col < ncols; ++col)
   for( int row = 0; row < nrows; ++row)
     a[row*ncols+col] += 1;
```