

# Paging: Faster Translations



**Operating Systems**

Baochun Li

University of Toronto

# Paging: Challenges

## Two major challenges —

1. The mapping from virtual to physical address must be **fast**

2. If the virtual address space is large, the linear page table will be large

Solved with multi-level page tables

# Challenge: Performance

**Remember that dynamic address translation occurs on the fly at run-time**

**Too slow, each memory access requires at least two — and up to six!**

**Solution?**

# Solution: Basic Idea

**MMU caches page table entries and knows how to handle cache misses**

**This cache is called the Translation Lookaside Buffer (TLB)**

# Translation Lookaside Buffer

# TLB Entries

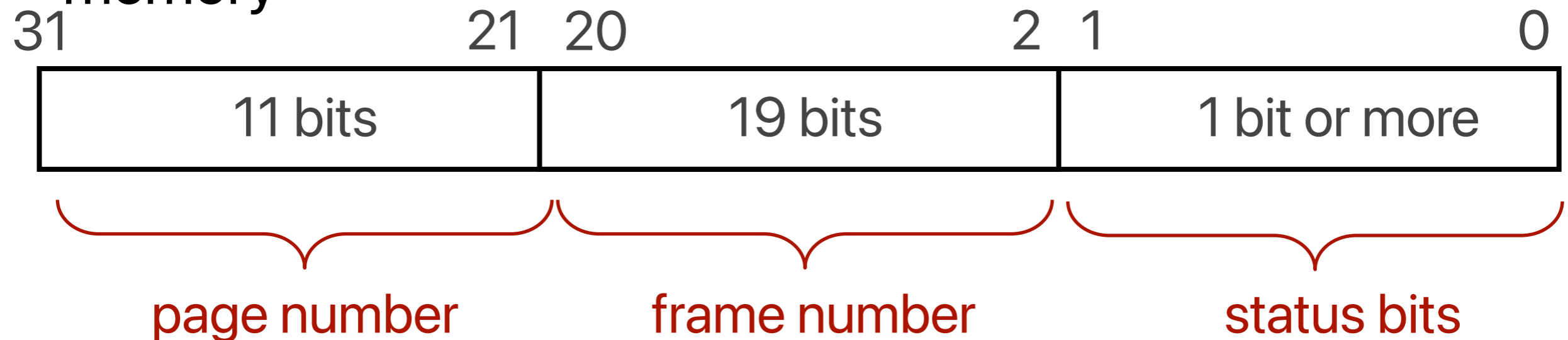
## Each entry contains

Page number (why do we need the page number here?)

Frame number

Valid bit

Other status bits from a page table entry in the physical memory



# TLB Operations

## Similar to operations on any cache

### Page lookup

Returns the frame number

Fully associative memory, indexed by page number

Performed in hardware, requires just a single cycle!

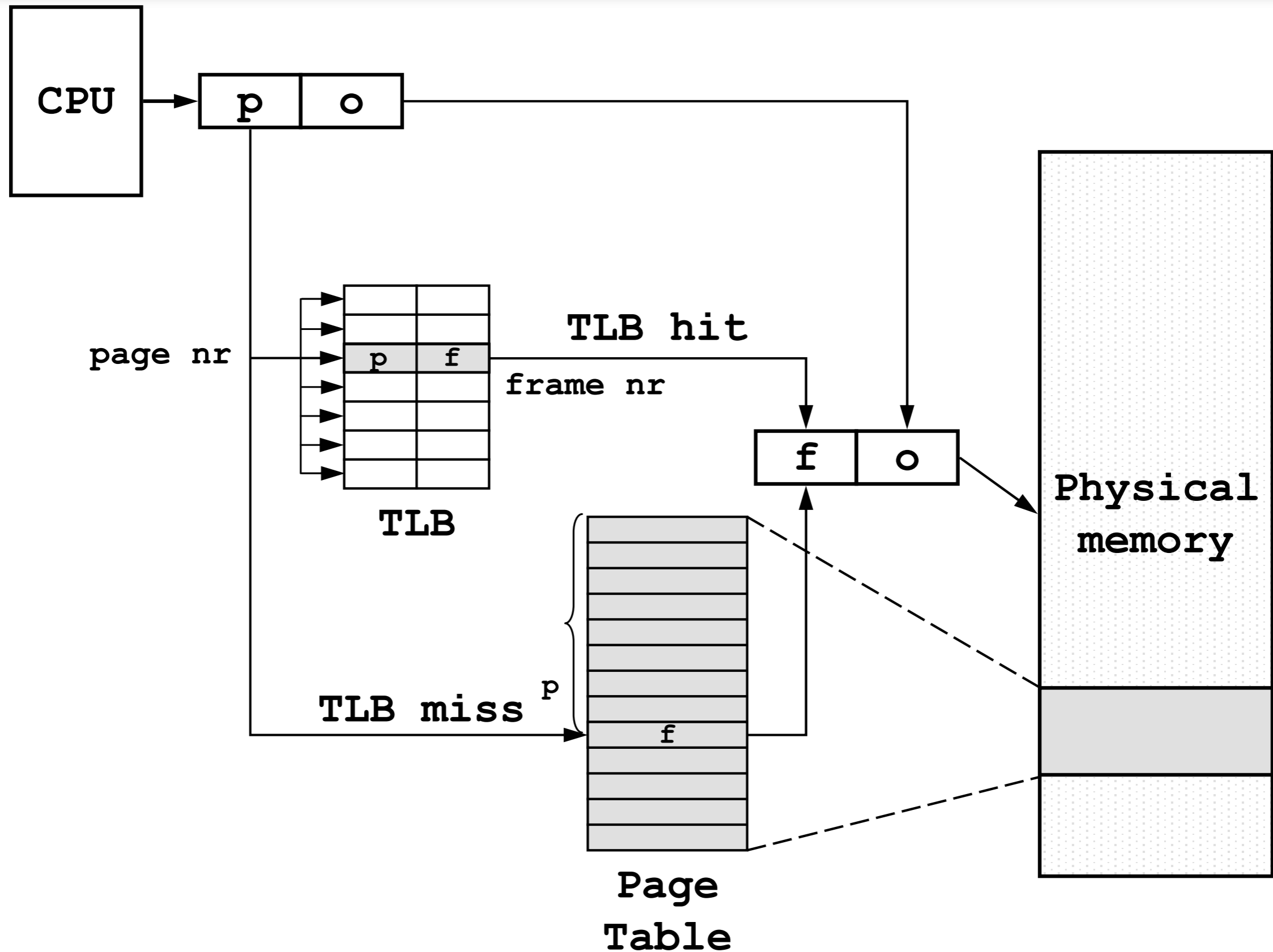
### Handling TLB misses

Lookup the page table for correct entry, fill TLB cache

### Invalidating stale TLB entries

If a page table entry is changed, the cached TLB entry must be invalidated or updated

# Page Lookup with a TLB





# TLB Control Flow

```
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()
```

# Handling TLB Misses

**TLB misses — What if the entry is not in the TLB?**

**Look in the page table in memory**

Find the right entry

Move it into the TLB

Which TLB entry should be replaced?

This is called the **TLB replacement policy**

# Handling TLB Misses

## Hardware Managed TLB (x86 CPUs)

TLB misses are handled in hardware

Hardware defines page table format, and uses the page table base register to locate the page table in physical memory

TLB replacement policy fixed by hardware

## Software Managed TLB (typical RISC CPUs: SPARC, MIPS)

Hardware generates an exception called **TLB miss fault**

OS handles TLB miss, similar to interrupt handling

The exception handler retrieves the correct page table entry, and adds it to the TLB

Replacement policy managed in software

# Invalidating TLB Entries

## When should a TLB entry be invalidated?

On a context switch to another thread in a different address space

### Why?

Prevents use of mapping in the previous address space

## Option 1

Empty the TLB, by clearing the valid bit of all entries

New thread will generate misses until it caches enough of its own entries into the TLB

## Option 2

Hardware maintains an "address-space ID" tag in each TLB entry

Hardware compares this tag to the current address space identifier, held in a specific register, on every translation

Enables space multiplexing, no need to invalidate all entries

# Enforcing Page-Level Protection

**We mentioned that page-level protection can be enforced during dynamic address translation with protection bits**

## **Option 1**

Check the page table on **each** memory access — Slow

## **Option 2**

Cache page-level protection bits in the TLB

Check TLB on **each** memory access — Fast

# Using TLB to Enforce Protection

**TLB checks whether memory accesses are valid when performing translation**

**If the memory access is of an invalid type (e.g., a page in the text segment is being modified), generate a **protection fault****

# What We've Covered So Far

## Three Easy Pieces: Chapter 19 (Paging: Faster Translations)