

# UC Berkeley CS61C : Machine Structures

## Lecture 35 – Virtual Memory II

2007-04-16

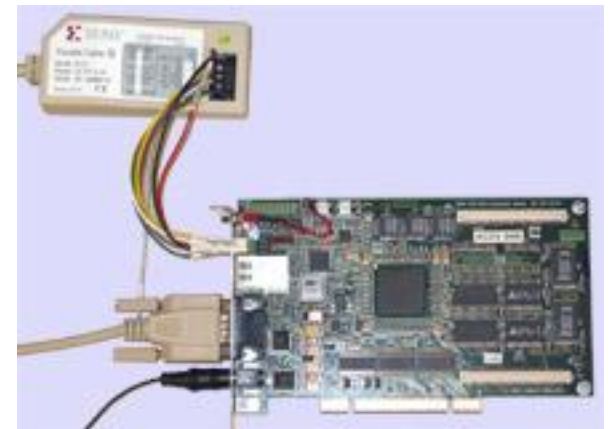


Lecturer SOE Dan Garcia

[www.cs.berkeley.edu/~ddgarcia](http://www.cs.berkeley.edu/~ddgarcia)

**Hardware repair?! ⇒**

This technology allows you to “patch” your hardware after it has been installed via “Pheonix” - FPGA (field programmable gate array). The bad news: hardware folks can be sloppy & fix later!



# Review

---

- **Manage memory to disk? Treat as cache**
  - **Included protection as bonus, now critical**
  - **Use Page Table of mappings for each process vs. tag/data in cache**
- **Virtual Memory allows protected sharing of memory between processes**
- **Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well**



# Comparing the 2 levels of hierarchy

---

**Cache version**

**Virtual Memory vers.**

**Block or Line**

**Page**

**Miss**

**Page Fault**

**Block Size: 32-64B**

**Page Size: 4K-8KB**

**Placement:  
Direct Mapped,  
N-way Set Associative**

**Fully Associative**

**Replacement:  
LRU or Random**

**Least Recently Used  
(LRU)**

**Write Thru or Back**

**Write Back**



# Notes on Page Table

---

- Solves Fragmentation problem: all chunks same size, so all holes can be used
- OS must reserve “**Swap Space**” on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory

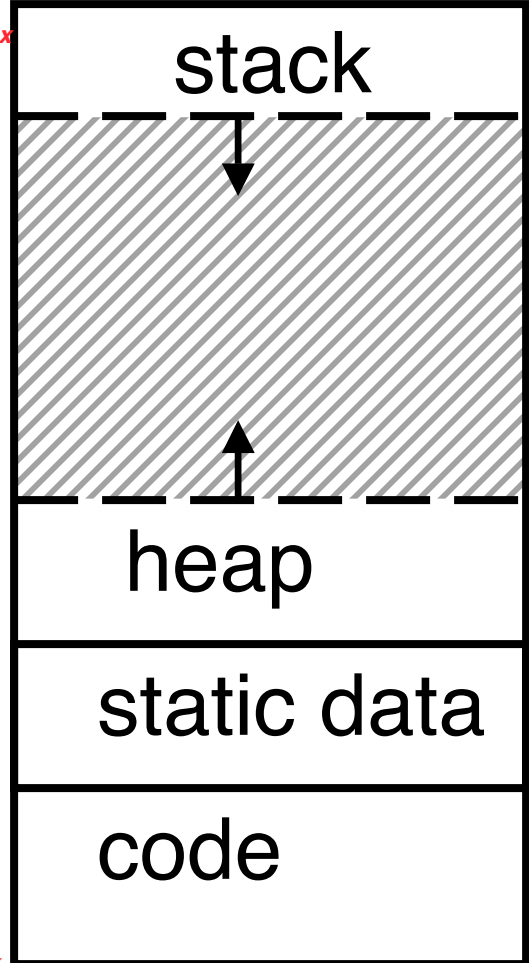


# Why would a process need to “grow”?

- A program’s *address space* contains 4 regions:
  - **stack**: local variables, **grows downward**
  - **heap**: space requested for pointers via `malloc()` ; resizes dynamically, **grows upward**
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change

*~ FFFF FFFF<sub>hex</sub>*

*~ 0<sub>hex</sub>*



*For now, OS somehow prevents accesses between stack and heap (gray hash lines).*



# Virtual Memory Problem #1

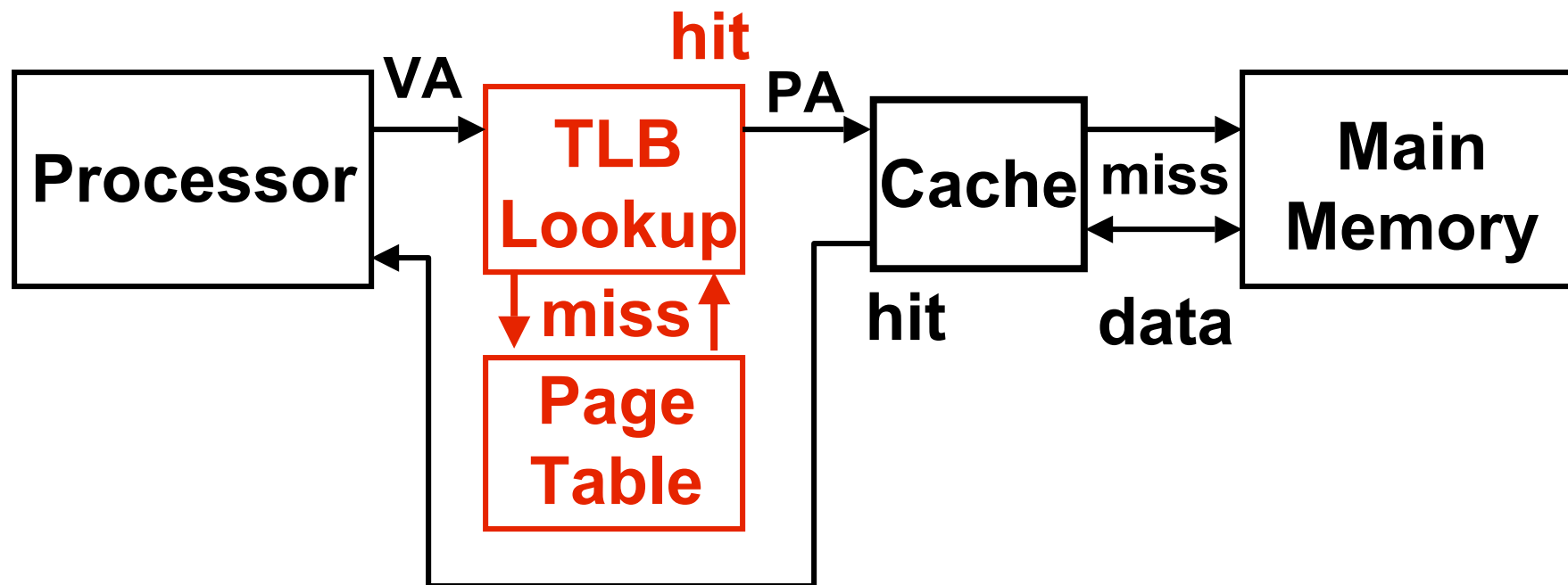
---

- Map every address  $\Rightarrow$  1 indirection via Page Table in memory per virtual address  $\Rightarrow$  1 virtual memory accesses = 2 physical memory accesses  $\Rightarrow$  SLOW!
- Observation: since locality in pages of data, there must be locality in virtual address translations of those pages
- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?
- For historical reasons, cache is called a Translation Lookaside Buffer, or TLB



# Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative

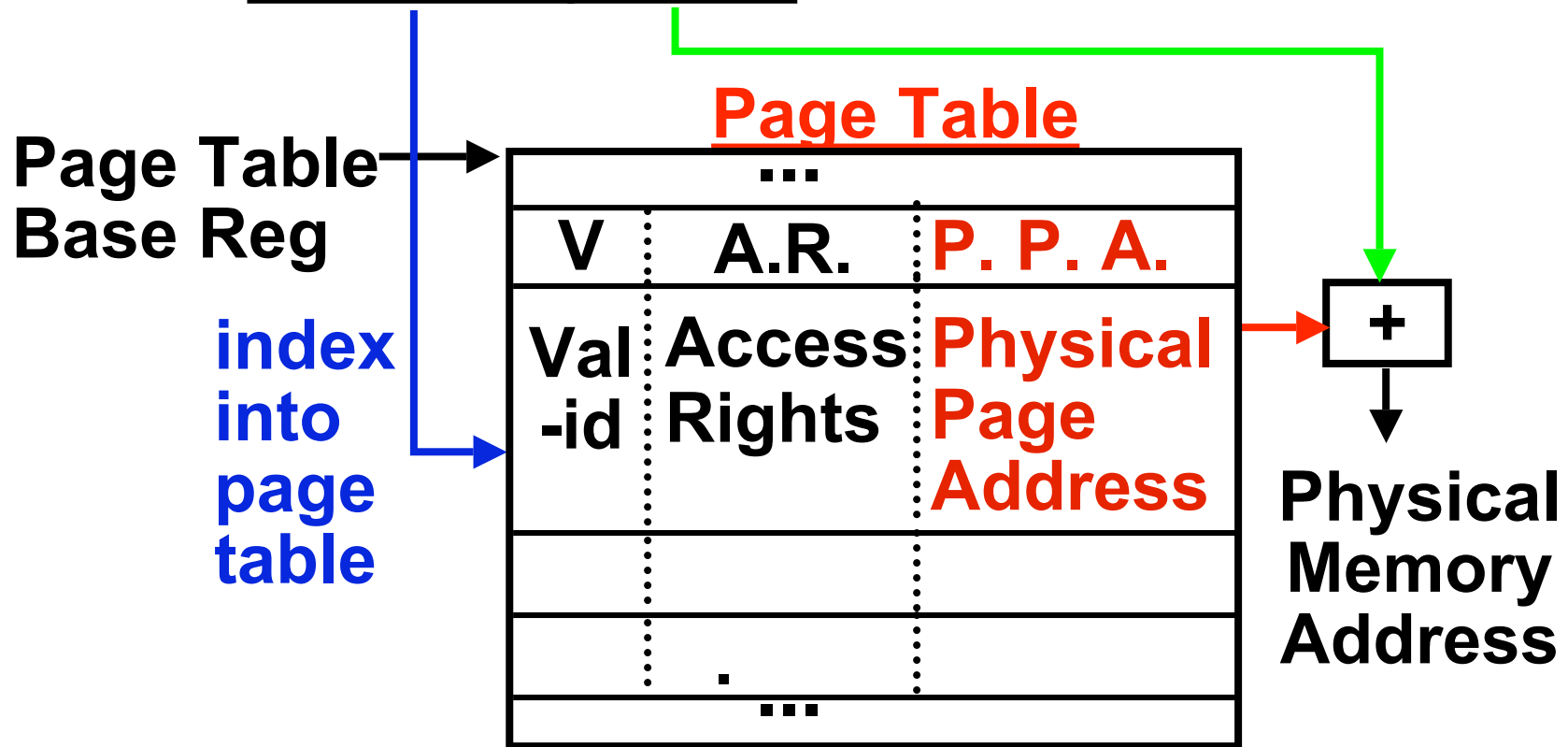


On TLB miss, get page table entry from main memory



# Review Address Mapping: Page Table

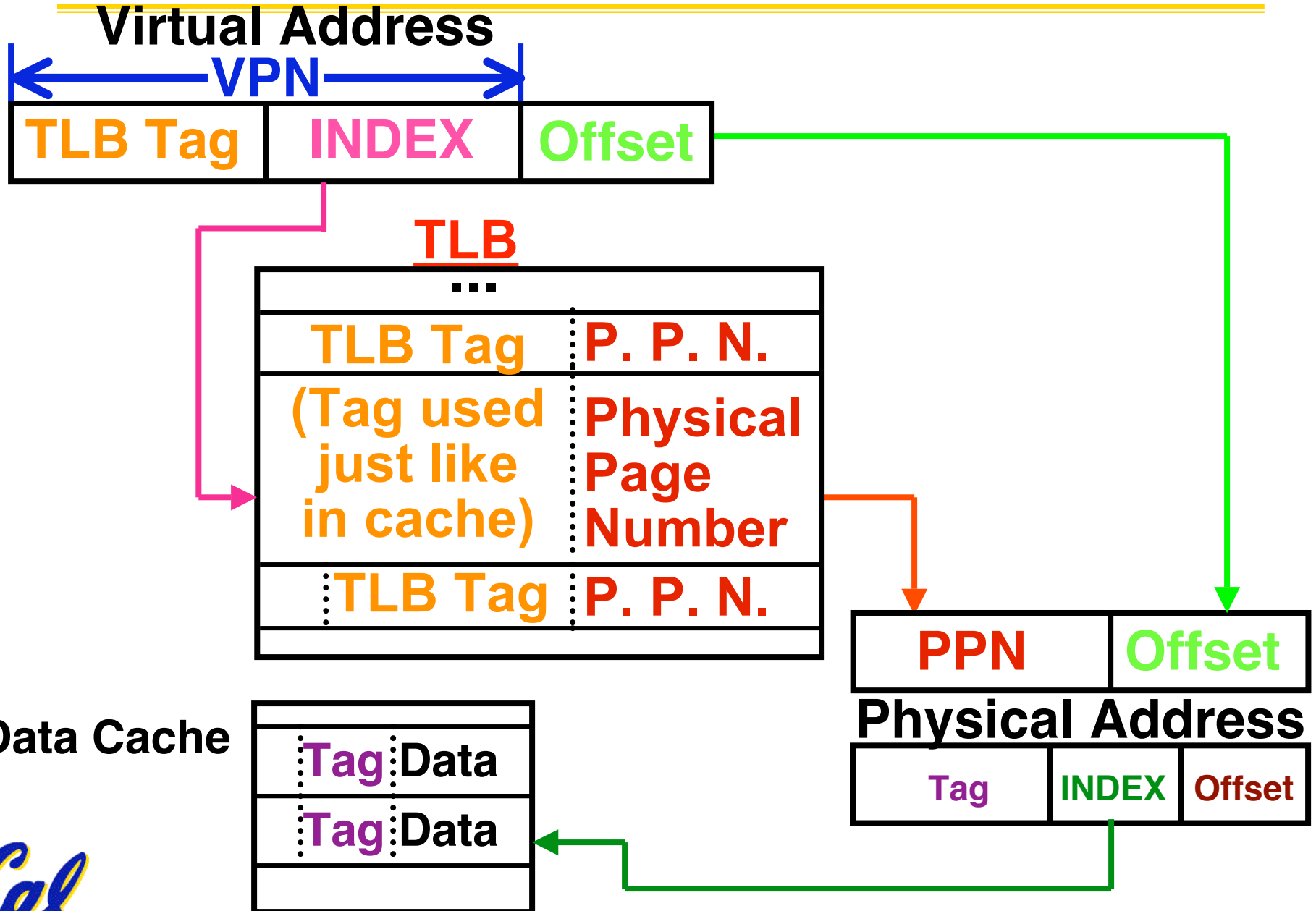
Virtual Address:



Page Table located in physical memory



# Address Translation using TLB



# Typical TLB Format

---

Tag	Physical Page #	Dirty	Ref	Valid	Access Rights

- TLB just a cache on the page table mappings
- TLB access time comparable to cache (much less than main memory access time)
- **Dirty**: since use write back, need to know whether or not to write page to disk when replaced
- **Ref**: Used to help calculate LRU on replacement
  - Cleared by OS periodically, then checked to see if page was **referenced**



# What if not in TLB?

---

- **Option 1: Hardware checks page table and loads new Page Table Entry into TLB**
- **Option 2: Hardware traps to OS, up to OS to decide what to do**
  - **MIPS follows Option 2: Hardware knows nothing about page table**



## What if the data is on disk?

---

- We load the page off the disk into a free block of memory, using a DMA transfer (Direct Memory Access – special hardware support to avoid processor)
  - Meantime we switch to some other process waiting to be run
- When the DMA is complete, we get an **interrupt** and update the process's page table
  - So when we switch back to the task, the desired data will be in memory



# What if we don't have enough memory?

---

- We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  - If clean (disk copy is up-to-date), just overwrite that data in memory
  - We chose the page to evict based on replacement policy (e.g., LRU)
- And update that program's page table to reflect the fact that its memory moved somewhere else
- If continuously swap between disk and memory, called **Thrashing**



---

**We're done with new material**

**Let's now review w/Questions**



# Peer Instruction

---

- A. **Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM**
- B. **Cache management is done by hardware (HW), page table management by the operating system (OS), but TLB management is either by HW or OS**
- C. **VM helps both with security and cost**

	ABC
0:	<b>FFF</b>
1:	<b>FFT</b>
2:	<b>FTF</b>
3:	<b>FTT</b>
4:	<b>TFF</b>
5:	<b>TFT</b>
6:	<b>TTF</b>
7:	<b>TTT</b>



## Question (1/3)

---

- 40-bit virtual address, 16 KB page

Virtual Page Number (? bits)	Page Offset (? bits)
------------------------------	----------------------

- 36-bit physical address

Physical Page Number (? bits)	Page Offset (? bits)
-------------------------------	----------------------

- Number of bits in Virtual Page Number/ Page offset, Physical Page Number/Page offset?

1: 22/18 (VPN/PO), 22/14 (PPN/PO)  
2: 24/16, 20/16  
3: 26/14, 22/14  
4: 26/14, 26/10  
5: 28/12, 24/12





## (1/3) Answer

- 40- bit virtual address, 16 KB ( $2^{14}$  B)

Virtual Page Number (26 bits)	Page Offset (14 bits)
-------------------------------	-----------------------

- 36- bit virtual address, 16 KB ( $2^{14}$  B)

Physical Page Number (22 bits)	Page Offset (14 bits)
--------------------------------	-----------------------

- Number of bits in Virtual Page Number/ Page offset, Physical Page Number/Page offset?

1: 22/18 (VPN/PO), 22/14 (PPN/PO)

2: 24/16, 20/16

3: 26/14, 22/14

4: 26/14, 26/10

5: 28/12, 24/12



## Question (2/3): 40b VA, 36b PA

- 2-way set-assoc. TLB, 512 entries, 40b VA:



- TLB Entry: Valid bit, Dirty bit, Access Control (say 2 bits), Virtual Page Number, Physical Page Number



- Number of bits in TLB Tag / Index / Entry?

1: 12 / 14 / 38 (TLB Tag / Index / Entry)  
2: 14 / 12 / 40  
3: 18 / 8 / 44  
4: 18 / 8 / 58



## (2/3) Answer

- 2-way set-associative data cache, 256 ( $2^8$ ) “sets”, 2 TLB entries per set  $\Rightarrow$  8 bit index



Virtual Page Number (26 bits)

- TLB Entry: Valid bit, Dirty bit, Access Control (2 bits), Virtual Page Number, Physical Page Number



1: 12 / 14 / 38 (TLB Tag / Index / Entry)

2: 14 / 12 / 40

3: 18 / 8 / 44

4: 18 / 8 / 58



## Question (3/3)

- 2-way set-assoc, 64KB data cache, 64B block



Physical Page Address (36 bits)

- Data Cache Entry: Valid bit, Dirty bit, Cache tag + ? bits of Data



- Number of bits in Data cache Tag / Index / Offset / Entry?

1: 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry)

2: 20 / 10 / 6 / 86

3: 20 / 10 / 6 / 534

4: 21 / 9 / 6 / 87

5: 21 / 9 / 6 / 535



## (3/3) Answer

- 2-way set-assoc data cache, 64K/1K ( $2^{10}$ ) “sets”, 2 entries per sets  $\Rightarrow$  9 bit index



Physical Page Address (36 bits)

- Data Cache Entry: Valid bit, Dirty bit, Cache tag + 64 Bytes of Data



- 1: 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry)
- 2: 20 / 10 / 6 / 86
- 3: 20 / 10 / 6 / 534
- 4: 21 / 9 / 6 / 87
- 5: 21 / 9 / 6 / 535



## And in Conclusion...

---

- **Virtual memory to Physical Memory Translation too slow?**
  - **Add a cache of Virtual to Physical Address Translations, called a TLB**
- **Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well**
- **Virtual Memory allows protected sharing of memory between processes with less swapping to disk**



# Bonus slides

---

- **These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.**
- **The slides will appear in the order they would have in the normal presentation**

# Bonus



# 4 Qs for any Memory Hierarchy

---

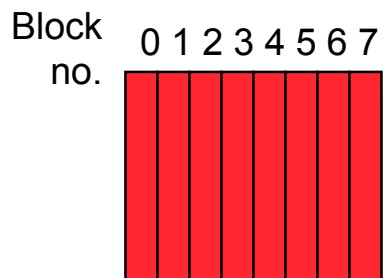
- **Q1: Where can a block be placed?**
  - One place (direct mapped)
  - A few places (set associative)
  - Any place (fully associative)
- **Q2: How is a block found?**
  - Indexing (as in a direct-mapped cache)
  - Limited search (as in a set-associative cache)
  - Full search (as in a fully associative cache)
  - Separate lookup table (as in a page table)
- **Q3: Which block is replaced on a miss?**
  - Least recently used (LRU)
  - Random
- **Q4: How are writes handled?**
  - Write through (Level never inconsistent w/lower)
  - Write back (Could be “dirty”, must have dirty bit)



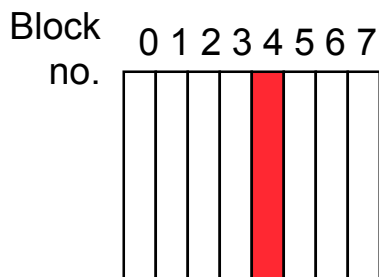


# Q1: Where block placed in upper level?

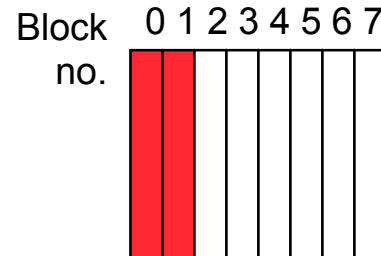
- **Block #12 placed in 8 block cache:**
  - Fully associative
  - Direct mapped
  - 2-way set associative
    - **Set Associative Mapping = Block # Mod # of Sets**



Fully associative:  
block 12 can go  
anywhere



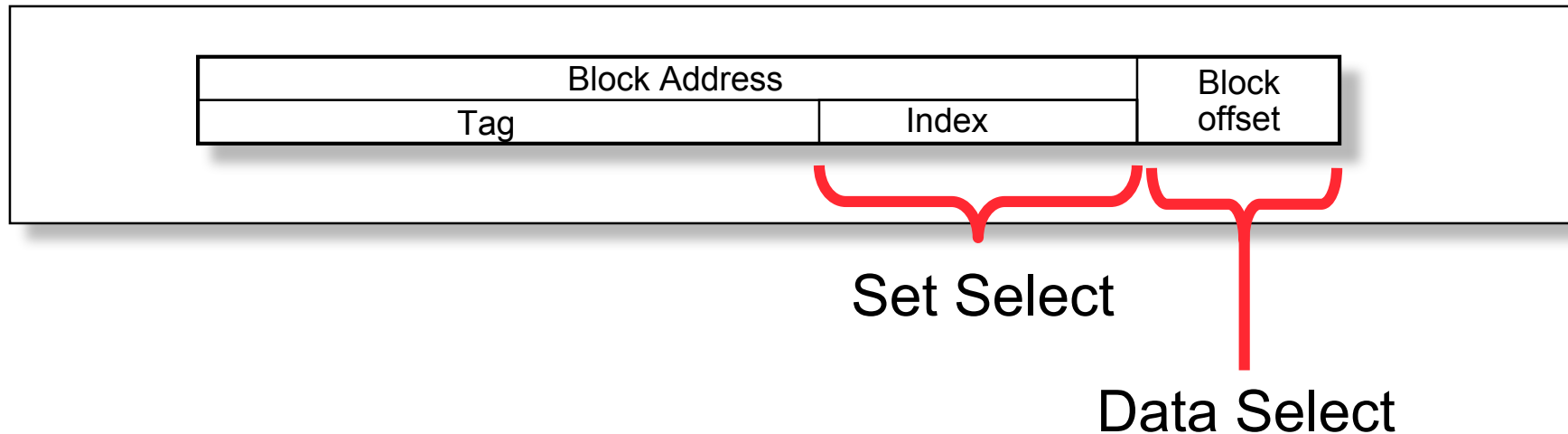
Direct mapped:  
block 12 can go  
only into block 4  
(12 mod 8)



Set Set Set Set  
0 1 2 3  
Set associative:  
block 12 can go  
anywhere in set 0  
(12 mod 4)



## Q2: How is a block found in upper level?



- **Direct indexing (using index and block offset), tag compares, or combination**
- **Increasing associativity shrinks index, expands tag**

## Q3: Which block replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

### Miss Rates

Size	Associativity:2-way		4-way		8-way	
	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%



## Q4: What to do on a write hit?

---

- Write-through

- update the word in cache block and corresponding word in memory

- Write-back

- update word in cache block
- allow memory word to be “stale”

=> add ‘dirty’ bit to each line indicating that memory be updated when block is replaced

=> OS flushes cache before I/O !!!

- Performance trade-offs?

- WT: read misses cannot result in writes

- WB: no writes of repeated writes



# Three Advantages of Virtual Memory

---

## 1) Translation:

- Program can be given consistent view of memory, even though physical memory is scrambled
- Makes multiple processes reasonable
- Only the most important part of program (“Working Set”) must be in physical memory
- Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later



# Three Advantages of Virtual Memory

---

## 2) Protection:

- Different processes protected from each other
- Different pages can be given special behavior
  - (Read Only, Invisible to user programs, etc).
- Kernel data protected from User programs
- Very important for protection from malicious programs ⇒ Far more “viruses” under Microsoft Windows
- Special Mode in processor (“Kernel mode”) allows processor to change page table/TLB

## 3) Sharing:

- Can map same physical page to multiple users (“Shared memory”)



# Why Translation Lookaside Buffer (TLB)?

---

- **Paging is most popular implementation of virtual memory (vs. base/bounds)**
- **Every paged virtual memory access must be checked against Entry of Page Table in memory to provide protection / indirection**
- **Cache of Page Table Entries (TLB) makes address translation possible without memory access in common case to make fast**



# Bonus slide: Virtual Memory Overview (1/4)

- **User program view of memory:**
  - Contiguous
  - Start from some set address
  - Infinitely large
  - Is the only running program
- **Reality:**
  - Non-contiguous
  - Start wherever available memory is
  - Finite size
  - Many programs running at a time





# Bonus slide: Virtual Memory Overview (2/4)

- **Virtual memory provides:**
  - illusion of contiguous memory
  - all programs starting at same set address
  - illusion of ~ infinite memory ( $2^{32}$  or  $2^{64}$  bytes)
  - protection



# **Bonus slide: Virtual Memory Overview (3/4)**

- **Implementation:**

- **Divide memory into “chunks” (pages)**
- **Operating system controls page table that maps virtual addresses into physical addresses**
- **Think of memory as a cache for disk**
- **TLB is a cache for the page table**



# Bonus slide: Virtual Memory Overview (4/4)

- **Let's say we're fetching some data:**
  - **Check TLB (input: VPN, output: PPN)**
    - **hit: fetch translation**
    - **miss: check page table (in memory)**
      - **Page table hit: fetch translation**
      - **Page table miss: page fault, fetch page from disk to memory, return translation to TLB**
  - **Check cache (input: PPN, output: data)**
    - **hit: return value**
    - **miss: fetch value from memory**



# Address Map, Mathematically

$V = \{0, 1, \dots, n - 1\}$  virtual address space ( $n > m$ )

$M = \{0, 1, \dots, m - 1\}$  physical address space

MAP:  $V \rightarrow M \cup \{\theta\}$  address mapping function

MAP( $a$ ) =  $a'$  if data at virtual address  $a$  is present in physical address  $a'$  and  $a'$  in  $M$   
=  $\theta$  if data at virtual address  $a$  is not present in  $M$

