Virtual Address Translation
Write-Allocate vs. Write-no-Allocate

Two options if a store causes a miss in the cache

- **Write-allocate**: Fetch line into cache, then perform the write in the cache
- **Write-no-allocate**: Pass the write through to the main memory, don’t bring the line into the cache

Tradeoffs

- WA has Better performance if data referenced again before it is evicted
- WNA has
  - Simpler write hardware
  - May be better for small caches if written data won’t be read again soon

Which makes more sense for writeback/write-through?
Instruction and Data Caches

CPU

References to Instructions
Instruction Cache (I-Cache)

References to Data
Data Cache (D-Cache)

Main Memory
Why Do We Do This?

- **Bandwidth**: lets us access instructions and data in parallel
- Most programs don’t modify their instructions
- I-Cache can be simpler than D-Cache, since instruction references are never writes
- Instruction stream has high locality of reference, can get higher hit rates with small cache
  - Data references never interfere with instruction references
Cache Performance Example

- **Given**
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions

What is the actual CPI?
## 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong> (per core)</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified cache</strong> (per core)</td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td><strong>L3 unified cache</strong> (shared)</td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
</tbody>
</table>

n/a: data not available
Microarchitecture of Cache Memories

Address

Tag Array

Data Array

Hit? Hit?
Why This Organization?

- Allows tag array to be faster than data array
  - Tag array is smaller
- Don’t really need output of data array until hit/miss detection complete
- Overlap some of data array access time with hit/miss detection
- Also integrates well with virtual memory, as we’ll see
Virtual Memory

- Virtual memory – separation of logical memory from physical memory.
  - Only a part of the program needs to be in memory for execution. Hence, logical address space can be much larger than physical address space.
  - Allows address spaces to be shared by several processes (or threads).
  - Allows more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Main memory is like a cache to the hard disc!
The concept of a virtual (or logical) address space that is bound to a separate physical address space is central to memory management.

- Virtual address: generated by the CPU
- Physical address: seen by the memory

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes; virtual and physical addresses differ in execution-time address-binding schemes.
Advantages of Virtual Memory

Translation:

- Program can be given consistent view of memory, even though physical memory is scrambled
- 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 grow later
Advantages of Virtual Memory

- **Protection:**
  - Different threads (or 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

- **Sharing:**
  - Can map same physical page to multiple users
    ("Shared memory")
Use of Virtual Memory

- Stack
- Shared Libs
- Heap
- Static data
- Code

Process A

- Stack
- Shared Libs
- Heap
- Static data
- Code

Process B

- Stack
- Shared Libs
- Heap
- Static data
- Code
Virtual vs. Physical Address Space

Virtual Address | Virtual Memory | Physical Address | Main Memory
--- | --- | --- | ---
0 | A | 0 | C
4k | B | 4k | D
8k | C | 8k | A
12k | D | 12k | B
| | | 16k | |
| | | 20k | |
| | | 24k | |
| | | 28k | |
4G | | | |

Disk

Main Memory

Physical Address

Virtual Address

Virtual Memory
Mapping Virtual to Physical Address

Virtual Address

31 30 29 28 27 ..................12 11 10
9 8 .................. 3 2 1 0

Translation

Physical Address

29 28 27 ..................12 11 10
9 8 .................. 3 2 1 0

1KB page size
Design considerations

- Main memory is about $100,000 \times$ faster than disk
  - Pages should be large enough to try to amortize the high access time
    - 4KB to 16MB in real systems

- Reduce page fault rate!
  - Allow fully associative placement of pages in memory

- Page faults can be handled in software
  - Can use clever algorithms for choosing how to place pages

- Write-through will not work well for VM
  - Use write-back
How to search the fully associative pages?

- A full search is impractical
- Use a table, page table, that indexes the memory

Page table

- The page table is indexed with the page number from the virtual address to find the corresponding physical page number
- To indicate the location of the page table, the page table register points to the start of the page table
- Each program has its own page table
  - Provide protection of one program from another
Translation w/ Single-Level Page Table

Virtual Address

VPN
Offset

Size of Page Table Entry

PTBR

PTBR

PTE

VPN

PPN

Memory Access

V

PTE

1

PPN

Physical Address

PPN
Offset

page table base register
page table entry
virtual page number
physical page number

PTBR

PhysAddr of PTE

PTE

PPN

VPN

Offset

Physical Address
Page Table Structure Examples

- One-to-one mapping, space?
  - Large pages → Internal fragmentation (similar to having large line sizes in caches)
  - Small pages → Page table size issues

Example:
64 bit address space, 4 KB pages (12 bits), 512 MB (29 bits) RAM

Number of pages = \(2^{64}/2^{12} = 2^{52}\)
(The page table has as many entrees)

Each entry is \(~4\) bytes, the size of the Page table is \(2^{54}\) Bytes = 16 Petabytes!

Can’t fit the page table in the 512 MB RAM!
TLB – a Cache for Page Table Entries

Virtual Address

VPN  Offset

TLB

V  D  VPN  PPN

Hit?

Yes

PPN  Offset

Physical Address

No

Use Page Table
Typical values for a TLB

- TLB size: 16-512 entries
- Block size: 1-2 page table entries (4-8 bytes each)
- Hit time: 0.5-1 clock cycle
- Miss penalty: 10-100 clock cycles
- Miss rate: 0.01-1%
Caches and Virtual Memory

- Do we send virtual or physical addresses to the cache?
  - Virtual $\rightarrow$ faster, because don’t have to translate
    - Issue: Different programs can reference the same virtual address, either creates security/correctness hole or requires flushing the cache every time you context switch
  - Physical $\rightarrow$ slower, but no security issue

- Actually, there are four possibilities
  - VIVT: Virtually-indexed Virtually-tagged Cache
  - PIPT: Physically-indexed Physically-tagged Cache
  - VIPT: Virtually-indexed Physically-tagged Cache
  - PIVT: Physically-indexed Virtually-tagged Cache
Virtually Indexed, Virtually Tagged

- Fast cache access
  - Only require address translation upon miss

- Issues
  - Homonym
    - Same VA maps to different PAs upon context switch
  - Synonym (also a problem in VIPT)
    - Different VAs map to the same PA when data is shared by multiple processes

```
Processor Core  
   VIVT Cache  
      TLB  
        Main Memory

VA  
   VA
hit

miss

cache line return
```
Physically-Indexed Physically-Tagged

- Slower, always translate address before accessing memory
- Simpler for data coherence
Virtually-Indexed Physically-Tagged

- Gain benefit of a VIVT and PIPT
  - Very common in commercial processors
  - Parallel Access to TLB and VIPT cache

- Issues
  - Synonym as VIVT; no homonym
DECStation 3100/MIPS R2000

Virtual Address

Virtual page number | Page offset

20 | 12

31 30 29 ............... 15 14 13 12 11 10 9 8 ........ 3 2 1 0

TLB

64 entries, fully associative

Valid Dirty | Tag | Physical page number

Physical Address

Physical page number | Page offset

16 | 14 | 2 offset

Physical address tag | Cache index

Cache

16K entries, direct mapped

Valid | Tag | Data

Data

Cache hit

Cache hit

32
Putting it all together
Another example

- Virtual Memory
  - Address width: 32 bits
  - Page size: 1 K bytes
- Single level page table
- Physical Memory
  - 32 bit physical address space
- Cache
  - Block size: 16 bytes
  - Cache size: 1 K bytes
  - Associativity: Direct mapped
- Translation Lookaside Buffer
  - Number of translations: 64
  - Associativity: Direct mapped
## 2-Level TLB Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual addr</td>
<td>48 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Physical addr</td>
<td>44 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>4KB, 2/4MB</td>
<td>4KB, 2/4MB</td>
</tr>
</tbody>
</table>
| L1 TLB (per core)      | L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages  
                          | L1 D-TLB: 64 entries for small pages, 32 for large pages  
                          | Both 4-way, LRU replacement         | L1 I-TLB: 48 entries  
                          | L1 D-TLB: 48 entries  
                          | Both fully associative, LRU replacement |
| L2 TLB (per core)      | Single L2 TLB: 512 entries  
                          | 4-way, LRU replacement               | L2 I-TLB: 512 entries  
                          | L2 D-TLB: 512 entries  
                          | Both 4-way, round-robin LRU         |
| TLB misses             | Handled in hardware                   | Handled in hardware                  |
## 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong> (per core)</td>
<td><strong>L1 I-cache</strong>: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td><strong>L1 I-cache</strong>: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td><strong>L1 D-cache</strong>: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td><strong>L1 D-cache</strong>: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified cache</strong> (per core)</td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td><strong>L3 unified cache</strong> (shared)</td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
</tbody>
</table>

n/a: data not available
Next lecture

- Pipelining

- References:
  - Chapter 5.4