C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly

get_mpg:
  pushq  %rbp
  movq   %rsp, %rbp
  ...
  popq   %rbp
  ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000001111110100001111

OS:

Virtual memory
Memory allocation
Java vs. C

Winter 2016
Virtual Memory (VM)

- Overview and motivation
- VM as tool for caching
- Address translation
- VM as tool for memory management
- VM as tool for memory protection
Again: Processes

- **Definition:** A *process* is an instance of a running program
  - One of the most important ideas in computer science
  - Not the same as “program” or “processor”

- **Process provides each program with two key abstractions:**
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private virtual address space
    - Each process seems to have exclusive use of main memory

- **How are these illusions maintained?**
  - Process executions interleaved (multi-tasking) – done...
  - Address spaces managed by virtual memory system – now!
Memory as we know it so far... is virtual!

- Programs refer to *virtual* memory addresses
  - `movq (%rdi), %rax`
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - System provides address space private to particular “process”

- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space

- But...
  - We probably don’t have exactly $2^w$ bytes of physical memory.
    - We definitely do not have $2^{64}$ bytes of physical memory.
  - We *certainly* don’t have $2^w$ bytes of physical memory *for every process*.
  - We have multiple processes that usually should not interfere with each other, but sometimes should share code or data
Problem 1: How Does Everything Fit?

64-bit virtual addresses can address several exabytes (18,446,744,073,709,551,616 bytes)

1 virtual address space per process, with many processes...

Physical main memory offers a few gigabytes (e.g. 8,589,934,592 bytes)

(Actually, physical memory is smaller than the period at the end of this sentence compared to the virtual address space.)
Problem 2: Memory Management

We have multiple processes:

Process 1
Process 2
Process 3
...
Process n

Each process has...

stack
heap
.text
.data
...

Physical main memory

What goes where?
Problem 3: How To Protect

Physical main memory

Process i

Process j

Problem 4: How To Share?

Physical main memory

Process i

Process j
How can we solve these problems?

- Fitting a huge address space into a tiny physical memory
- Managing the address spaces of multiple processes
- Protecting processes from stepping on each other’s memory
- Allowing processes to share common parts of memory
Indirection

“Any problem in computer science can be solved by adding another level of indirection.” –David Wheeler, inventor of the subroutine (a.k.a. procedure)

- Without Indirection
  - Name \rightarrow \text{Thing}

- With Indirection
  - Name \rightarrow \text{Indirection} \rightarrow \text{Thing}

What if I want to move Thing?
Indirection

- **Indirection**: the ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.

- **Without Indirection**

- **With Indirection**

- **Examples of indirection:**
  - Domain Name Service (DNS): translation from name to IP address
  - phone system: cell phone number portability
  - snail mail: mail forwarding
  - 911: routed to local office
  - Dynamic Host Configuration Protocol (DHCP): local network address assignment
  - call centers: route calls to available operators, etc.
Each process gets its own private virtual address space
Solves the previous problems
Address Spaces

- **Virtual address space:** Set of $N = 2^n$ virtual addresses
  \{0, 1, 2, 3, ..., N-1\}

- **Physical address space:** Set of $M = 2^m$ physical addresses ($n \geq m$)
  \{0, 1, 2, 3, ..., M-1\}

- **Every byte in main memory has:**
  - one physical address
  - zero, one, *or more* virtual addresses
Mapping

A virtual address can be mapped to either physical memory or disk

P1’s Virtual Address Space

P2’s Virtual Address Space

Physical Memory

Disk
A System Using Physical Addressing

- Used in “simple” systems with (usually) just one process:
  - embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

- Physical addresses are completely invisible to programs
- Used in all modern desktops, laptops, servers, smartphones...
- One of the great ideas in computer science
Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed

- **Simplifies memory management for programmers**
  - Each process gets the same full, private linear address space

- **Isolates address spaces**
  - One process can’t interfere with another’s memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions
VM and the Memory Hierarchy

- Think of virtual memory as array of $N = 2^n$ contiguous bytes.
- Pages of virtual memory are usually stored in physical memory, but sometimes spill to disk.
  - Pages are another unit of aligned memory (size is $P = 2^p$ bytes)
  - Each virtual page can be stored in *any* physical page

Virtual memory

<table>
<thead>
<tr>
<th>VP 0</th>
<th>VP 1</th>
<th>VP $2^n-1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Cached</td>
<td>Uncached</td>
</tr>
<tr>
<td>Uncached</td>
<td>Unallocated</td>
<td>Cached</td>
</tr>
<tr>
<td>Cached</td>
<td>Uncached</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory

<table>
<thead>
<tr>
<th>PP 0</th>
<th>PP 1</th>
<th>PP $2^{m-p}$-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty</td>
<td></td>
</tr>
</tbody>
</table>

Virtual pages (VP's) stored on disk

Physical pages (PP's) cached in DRAM

Disk
or: Virtual Memory as DRAM Cache for Disk

- Think of virtual memory as an array of $N = 2^n$ contiguous bytes stored on a disk.
- Then physical main memory is used as a cache for the virtual memory array
  - The cache blocks are called pages (size is $P = 2^p$ bytes)

![Diagram showing virtual memory and physical memory relation]

Virtual memory

- VP 0
  - Unallocated
  - Cached
- VP 1
  - Uncached
- VP $2^{n-p}-1$
  - Unallocated
  - Cached

Physical memory

- PP 0
  - Empty
- PP 1
  - Empty
- PP $2^{m-p}-1$
  - Empty

Virtual pages (VPs) stored on disk

Physical pages (PPs) cached in DRAM
Memory Hierarchy: Core 2 Duo

Not drawn to scale

SRAM
Static Random Access Memory

- L1 I-cache
- L1 D-cache
- 32 KB

Throughput: 16 B/cycle
Latency: 3 cycles

Miss penalty (latency): 33x

L2 unified cache

~4 MB

Miss penalty (latency): 10,000x

DRAM
Dynamic Random Access Memory

- Main Memory
- ~4 GB

Throughput: 2 B/cycle
Latency: 100 cycles

Throughput: 1 B/30 cycles
Latency: millions

Disk

~500 GB
Virtual Memory Design Consequences

- Large *page* size: typically 4-8 KB, sometimes up to 4 MB
- Fully associative
  - Any virtual page can be placed in any physical page
  - Requires a “large” mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware
- Write-back rather than write-through
How do we perform the virtual -> physical address translation?
A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
Summary of Address Translation Symbols

- **Basic Parameters**
  - \( N = 2^n \): Number of addresses in virtual address space
  - \( M = 2^m \): Number of addresses in physical address space
  - \( P = 2^p \): Page size (bytes)

- **Components of the virtual address (VA)**
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number

- **Components of the physical address (PA)**
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
Address Translation With a Page Table

In most cases, the hardware (the MMU) can perform this translation on its own, without software assistance.
**Page Hit**

- **Page hit:** reference to VM byte that is in physical memory

![Diagram of page hit](image)

**Virtual address**

**Physical page number or disk address**

**Memory resident page table (DRAM)**

**Physical memory (DRAM)**

**Virtual memory (disk)**
Page Fault

- **Page fault**: reference to VM byte that is \textbf{NOT} in physical memory

What happens when a page fault occurs?
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:  c7 05 10 9d 04 08 0d  movl  $0xd,0x8049d10
```

- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed *again*!
- Successful on second try
Handling Page Fault

- Page miss causes page fault (an exception)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!
Why does it work?
Why does VM work on RAM/disk? Locality.

- Virtual memory works well for avoiding disk accesses because of locality
  - Same reason that L1 / L2 / L3 caches work

- The set of virtual pages that a program is “actively” accessing at any point in time is called its *working set*

- If (working set size of one process < main memory size):
  - Good performance for one process (after compulsory misses)

- But if
  - SUM(working set sizes of all processes) > main memory size:
    - *Thrashing*: Performance meltdown where pages are swapped (copied) between memory and disk continuously. CPU always waiting or paging.
Simplifying Linking and Loading

**Linking**
- Each program has similar virtual address space
- Code, data, and heap always start at the same addresses.

**Loading**
- `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system
VM for Managing Multiple Processes

- Key abstraction: each process has its own virtual address space
  - It can view memory as a simple linear array

- With virtual memory, this simple linear virtual address space need not be contiguous in physical memory
  - Process needs to store data in another VP? Just map it to any PP!

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:
VM for Protection and Sharing

- The mapping of VPs to PPs provides a simple mechanism to **protect** memory and to **share** memory between processes
  - **Sharing**: just map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection**: process simply can’t access physical pages to which none of its virtual pages are mapped (here: Process 2 can’t access PP 2)
Memory Protection Within a Single Process

- Can we use virtual memory to control read/write/execute permissions? How?
Memory Protection Within a Single Process

- Extend page table entries with permission bits
- MMU checks these permission bits on every memory access
  - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Physical Page Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Physical Page Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Physical Address Space
Terminology

- **context switch**
  - Switch between processes on the same CPU

- **page in**
  - Move pages of virtual memory from disk to physical memory

- **page out**
  - Move pages of virtual memory from physical memory to disk

- **thrash**
  - Total working set size of processes is larger than physical memory
  - Most time is spent paging in and out instead of doing useful computation
Address Translation: Page Hit

1) Processor sends virtual address to MMU (*memory management unit*)

2-3) MMU fetches PTE from page table in cache/memory
   (Uses PTBR to find beginning of page table for current process)

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

VA = Virtual Address
PTEA = Page Table Entry Address
PA = Physical Address
Data = Contents of memory stored at VA originally requested by CPU
Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in cache/memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Hmm... Translation Sounds Slow!

- The MMU accesses memory *twice*: once to first get the PTE for translation, and then again for the actual memory request from the CPU
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- *What can we do to make this faster?*
Speeding up Translation with a TLB

Solution: add another cache!

**Translation Lookaside Buffer (TLB):**
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete *page table entries* for small number of pages
  - Modern Intel processors: 128 or 256 entries in TLB
- Much faster than a page table lookup in cache/memory
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare.
Summary of Address Translation Symbols

- **Basic Parameters**
  - $N = 2^n$: Number of addresses in virtual address space
  - $M = 2^m$: Number of addresses in physical address space
  - $P = 2^p$: Page size (bytes)

- **Components of the virtual address (VA)**
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number

- **Components of the physical address (PA)**
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
Simple Memory System Example (small)

Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes
Simple Memory System Page Table

- Only showing first 16 entries (out of $2^8 = 256$)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>

- What about a real address space? Read more in the book...
Simple Memory System **TLB**

- 16 entries total
- 4 sets
- 4-way associative

TLB ignores page offset. Why?

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System **Cache**

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

Note: It is just random that the physical page number is the same bits as the cache tag
So...

- This seems complicated, but also elegant and effective
  - Level of indirection to provide isolated memory, caching, etc.
  - TLB as a cache-of-a-page-table to avoid “two trips to memory for one load”

- Just one issue... Numbers don’t work out for the story so far!

- The problem is the page-table itself for each process...
  - Suppose 64-bit addresses and 8KB pages
  - How many page-table-entries is that? (Also: Each PTE is > 1byte)
  - Moral: Cannot use this naïve implementation of the virtual->physical-page mapping: It’s way too big
A solution: Multi-level page tables

- **Page table base register (PTBR)**
- **VIRTUAL ADDRESS**
  - VPN 1
  - VPN 2
  - ... (Level 1 page table)
  - VPN k
  - VPO
- **PHYSICAL ADDRESS**
  - PPN
  - PPO

- **m-1**
- **n-1**
- **p-1**
This works!

- Just a tree of depth k (e.g., 4) where each node at depth i has up to $2^k$ children if part i of the VPN has k bits

- HW for multi-level page tables inherently more complicated, but it’s a necessary complexity: 1-level does not fit

- Why it works: Most subtrees are not used at all, so they are never created and definitely aren’t in physical memory
  - Even parts created can be evicted from cache/memory when not being used
  - Each node can have a size of ~1-100KB

- But now for a k-level page table, a TLB miss requires k+1 cache/memory accesses
  - Fine so long as TLB misses are rare: motivates larger TLBs
Summary

- **Programmer’s view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and sharing
  - Simplifies protection by providing a convenient interpositioning point to check permissions
Memory System Summary

- L1/L2 Memory Cache
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware

- Virtual Memory
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)
Memory System – Who controls what?

- L1/L2 Memory Cache
  - Controlled by hardware
  - Programmer cannot control it
  - Programmer *can* write code in a way that takes advantage of it

- Virtual Memory
  - Controlled by OS and hardware
  - Programmer cannot control mapping to physical memory
  - Programmer can control sharing and some protection
    - via OS functions (not in CSE 351)
Quick Review

- What do Page Tables Map?
- Where are Page Tables?
- How many Page Tables are there?
- Can your program tell if a page fault has occurred?
- What is thrashing?
- T/F Virtual Addresses that are contiguous will always be contiguous in physical memory.
- TLB stands for ________________ and stores ________________
Quick Review Answers

- What do Page Tables Map?
  - Virtual pages to physical pages or location on disk

- Where are Page Tables?
  - In physical memory

- How many Page Tables are there?
  - One per process

- Can your program tell if a page fault has occurred?
  - Nope. But it has to wait a long time.

- What is thrashing?
  - Constantly paging out and paging in. The working set of all applications you are trying to run is bigger than physical memory.

- T/F Virtual Addresses that are contiguous will always be contiguous in physical memory. (could be on different physical pages)

- TLB stands for Translation lookaside buffer, and stores page table entries
Detailed Examples...
# Current state of caches/tables

## TLB

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

## Cache

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
# Address Translation Example #1

**Virtual Address:** 0x03D4

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____**

**Physical Address**

<table>
<thead>
<tr>
<th>CT</th>
<th>CI</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**CO ___ CI___ CT ____ Hit? __ Byte: ____**
Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

CO 0  
CI 0x5  
CT 0x0D  
Hit? Y  
Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address

<table>
<thead>
<tr>
<th>CT</th>
<th>CI</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CO ___ CI___ CT ____ Hit? __ Byte: ____
Address Translation Example #2

Virtual Address: 0x0B8F

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

VPN ___ TLBI ___ TLBT ___ TLB Hit? ___ Page Fault? ___ PPN: TBD

VPN 0x2E TLBI 2 TLBT 0xB TLB Hit? N Page Fault? ? PPN: TBD

Physical Address

<table>
<thead>
<tr>
<th>CO</th>
<th>CI</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CO ___ CI ___ CT ___ Hit? ___ Byte: ___
Address Translation Example #3

Virtual Address: 0x0020

Virtual Memory

Physical Address
Address Translation Example #3

Virtual Address: 0x0020

Virtual Memory

Physical Address

CO 0
Cl 0x8
CT 0x28
Hit? N
Byte: Mem
Address Translation Example #4

Virtual Address: 0x036B

Physical Address

CO  CI  CT  Hit?  Byte:
Address Translation Example #4

Virtual Address: 0x036B

<table>
<thead>
<tr>
<th>TLBI</th>
<th>TLBT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

VPN: 0xD
TLB Hit? Y
Page Fault? N
PPN: 0x2D

Physical Address

<table>
<thead>
<tr>
<th>CO</th>
<th>CI</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0xA</td>
<td>0xD</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Hit? Y
Byte: 0x3B