Virtual Memory, Course Review

Pramod V. Argade
CSE141: Introduction to Computer Architecture

Instructor:  Pramod V. Argade (p2argade@cs.ucsd.edu)
Office Hours:
  Tue.  7:30 - 8:30 PM (AP&M 4141)
  Wed. 4:30 - 5:30 PM (AP&M 4141)

TA:
  Anjum Gupta (a3gupta@cs.ucsd.edu)
  Office Hour: Mon/Wed 12 - 1 PM
  Chengmo Yang (c5yang@cs.ucsd.edu)
  Office Hour: Mon/Thu 2 - 3 PM

Lecture:  Mon/Wed. 6 - 8:50 PM, Center 109

Textbook:  Computer Organization & Design
            Authors: Patterson and Hennessy

Web-page:  http://www-cse.ucsd.edu/classes/su04/cse141
Announcements

- **Pramod Argade:**
  - Extra Office Hour: Thursday, July 29, 4 - 5 PM

- **Anjum Gupta**
  - Special Midterm and Review Session: Thursday, 07/29/04 from 07:30 to 09:00 PM in CENTR 214

- **Reading Assignment:**
  - Virtual Memory, Section 7.4 - 7.5 (Wednesday)

- **Homework 6:** Due Fri., July 30 During Discussion
  - Cache: 7.7, 7.8, 7.9, 7.15, 7.16, 7.18, 7.20, 7.21
  - Virtual Memory: 7.32, 7.33

- **Final Exam**
  - **When:** Sat., July 31, 7 - 10 PM, **Center 101 (Note room change!)**
<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Time</th>
<th>Room</th>
<th>Topic</th>
<th>Quiz topic</th>
<th>Homework Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mon. 6/28</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Introduction, Ch. 1 ISA, Ch. 3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Wed. 6/30</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Performance, Ch. 2 Arithmetic, Ch. 4</td>
<td>ISA Ch. 3</td>
<td>#1</td>
</tr>
<tr>
<td></td>
<td>Mon. 7/5</td>
<td></td>
<td>No Class</td>
<td>July 4th Holiday</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Wed. 7/7</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Arithmetic, Ch. 4 Cont. Single-cycle CPU Ch. 5</td>
<td>Performance Ch. 2</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>Mon. 7/12</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Single-cycle CPU Ch. 5 Cont. Multi-cycle CPU Ch. 5</td>
<td>Arithmetic, Ch. 4</td>
<td>#3</td>
</tr>
<tr>
<td>5</td>
<td>Tue. 7/13</td>
<td>7:30 - 8:50 PM</td>
<td>Center 109</td>
<td>Multi-cycle CPU Ch. 5 Cont. (July 5th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Wed. 7/14</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Single and Multicycle CPU Examples and Review for Midterm</td>
<td>Single-cycle CPU Ch. 5</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Mon. 7/19</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Mid-term Exam Exceptions</td>
<td>-</td>
<td>#4</td>
</tr>
<tr>
<td>8</td>
<td>Tue. 7/20</td>
<td>7:30 - 8:50 PM</td>
<td>Center 109</td>
<td>Pipelining Ch. 6 (July 5th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Wed. 7/21</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Hazards, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Mon. 7/26</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Memory Hierarchy &amp; Caches Ch. 7</td>
<td>Hazards Ch. 6</td>
<td>#5</td>
</tr>
<tr>
<td>11</td>
<td>Wed. 7/28</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Virtual Memory, Ch. 7 Course Review</td>
<td>Cache Ch. 7</td>
<td>#6</td>
</tr>
<tr>
<td>12</td>
<td>Sat. 7/31</td>
<td>7 - 10 PM</td>
<td>Center 109</td>
<td>Final Exam</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Flexible Placement of Blocks

- **Direct mapped cache**
  - A block can go in exactly one place in the cache
  - Leads to collision among blocks

- **Fully associative cache**
  - A block can go in any place in the cache
  - All addresses have to be compared simultaneously
  - Slow and expensive

- **N-way set-associative cache**
  - Consists of a number of sets
  - Each set consists of N blocks
  - Each block in memory maps to a unique set
    - A block can be placed in any element of the set

**Set containing a memory block**

\[ \text{Set containing a memory block} = (\text{block number}) \mod (\text{number of sets in the cache}) \]

**Number of sets in the cache**

\[ \text{Number of sets in the cache} = \frac{\text{Cache size}}{((\text{block size}) \times \text{associativity})} \]
Locating a Block in a Cache

Block address = $12_{10}$

**Direct Mapped**
- Direct mapped
- Block # 0 1 2 3 4 5 6 7
- Tag
  - 1
  - 2
- Search

**Set-associative**
- Set associative
- Set # 0 1 2 3
- Tag
  - 1
  - 2
- Search

**Fully associative**
- Fully associative
- 1
- 2
- Search
Cache Configurations

- An eight-block cache with various configurations

On-way Associative
(direct mapped)

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eight-way set associative (fully associative)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Accessing a 4-way Set-associative cache

Number of Blocks
= Cache Size/Block Size
= 4 Kbytes/4 Bytes
= 1 K blocks

Number of Sets
= (# Blocks)/associativity
= 1 K/4
= 256

Index bits
= \( \log_2(\# \text{ Sets} ) \)
= \( \log_2(256) \)
= 8

• 4 K-byte 4-way set-associative cache, with a block size of 4 bytes
Accessing a Direct Mapped Cache

- 64 KB cache, direct-mapped, 32-byte cache block size

![Diagram of direct-mapped cache with tags, index, and data with valid bits]
Accessing a Set-associative Cache

- 32 KB cache, 2-way set-associative, 16-byte block size
A Fully-associative cache

address string:
20 00010100
5  00000101
10 00001010
12 00001100
  00001000
  00010001
  00001111
  00001000
  00010101
  00011000
  00001110
  00001011
  00000100

- A cache that can put a block of data anywhere is called **fully associative**
- To access the cache, address must be compared with all the entries in the cache
Cache Organization

- A typical cache has three dimensions

```
<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Number of sets (cache size)
- Blocks/set (associativity)
- Bytes/block (block size)

```
tag     index     block offset
```
The Three Cs

- **Compulsory misses**
  - Caused by the first access to a block that has never been in the cache
  - Also called cold-start misses
  - Can be reduced by increasing the block size

- **Capacity misses**
  - Caused when cache cannot contain all the blocks needed
  - Occur because of blocks being replaced and later retrieved
  - Can be reduced by enlarging the cache

- **Conflict misses**
  - Occur in direct mapped and set-associative caches
  - Multiple blocks compete for the same set
  - Can be eliminated by using fully associative cache
Which Block Should be Replaced on a Miss?

- Direct Mapped is Easy
- Set associative or fully associative:
  - Random (large associativities)
  - LRU (smaller associativities)
- Miss rates for the two schemes:

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 KB</td>
<td>LRU: 5.18%</td>
<td>Random: 5.69%</td>
<td>LRU: 4.67%</td>
</tr>
<tr>
<td>64 KB</td>
<td>LRU: 1.88%</td>
<td>Random: 2.01%</td>
<td>LRU: 1.54%</td>
</tr>
<tr>
<td>256 KB</td>
<td>LRU: 1.15%</td>
<td>Random: 1.17%</td>
<td>LRU: 1.13%</td>
</tr>
</tbody>
</table>

LRU is preferred scheme for a small size cache
Associative Caches: Higher hit rates, but...

- Longer access time (longer to determine hit/miss, more muxing of outputs)
- More space (longer tags)
  - 16 KB, 16-byte blocks, direct mapped, tag = ?
  - 16 KB, 16-byte blocks, 4-way, tag = ?
Summary

• The Principle of Locality:
  – Program likely to access a relatively small portion of the address space at any instant of time.
    • Temporal Locality: Locality in Time
    • Spatial Locality: Locality in Space

• Three Major Categories of Cache Misses:
  – Compulsory Misses: sad facts of life. Example: cold start misses.
  – Conflict Misses: increase cache size and/or associativity.
    Nightmare Scenario: ping pong effect!
  – Capacity Misses: increase cache size

• Cache Design Space
  – total size, block size, associativity
  – replacement policy
  – write-hit policy (write-through, write-back)

• Caches give an illusion of a large, cheap memory with the access time of a fast, expensive memory
Virtual Memory

- Virtual memory is the name of the technique that allows us to view main memory as a cache of a larger memory space (on disk).
  - Allows efficient and safe sharing of memory among programs
  - Creates an illusion of providing unlimited memory to programs
Virtual Memory (VM)

- Each program is compiled to run in its own address space
- A single program may exceed the size of primary memory
- Multiple programs may dynamically share portions of memory
- Main memory need contain only the active portions of the program
- VM and caching have different historical roots
- VM is like caching, but uses different terminology

<table>
<thead>
<tr>
<th>Cache</th>
<th>VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>page</td>
</tr>
<tr>
<td>cache miss</td>
<td>page fault</td>
</tr>
<tr>
<td>address</td>
<td>virtual address</td>
</tr>
<tr>
<td>index</td>
<td>physical address (sort of)</td>
</tr>
</tbody>
</table>
Advantages of Virtual Memory

- Performance
  - Large amount of memory accessed efficiently
- Memory sharing among multiple programs
- Protection
  - Simultaneous (time-sharing) execution of multiple programs
  - Use of “kernel space” and “user space”
- Ease of programming/compilation
- Efficient use of memory
Memory Mapping/Address Translation

- Virtual to physical address mapping
- Page may be present or absent in main memory
- Page may be resident on the disk
- Two virtual pages may map to the same physical address

![Diagram of memory mapping and address translation]
Mapping Virtual to Physical Address

Virtual Address

Virtual Page Number Page Offset

Page Table

Physical Page Number

Main Memory
Mapping from a Virtual to a Physical Address

Example:
- Virtual address space 4 Gbytes
- Physical address space 1 Gbyte
- Page size 4 Kbytes

Virtual Address

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12</td>
<td></td>
</tr>
<tr>
<td>11 10 9 8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Translation

<table>
<thead>
<tr>
<th>Physical page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 28 27</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12</td>
<td></td>
</tr>
<tr>
<td>11 10 9 8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Physical address
Address Translation via the Page Table

Notes:
• The page table contains mapping for every possible virtual page
• Valid bit indicates whether the page is present in the main memory
• Extra bits in the page table are used for protection information
Cache vs Virtual Memory Access

- Access time
time between when a read is requested and when the desired word arrives
- Transfer time
time it takes to transfer the whole request (ties up bandwidth)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>First-level Cache</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (Page) size</td>
<td>16 - 128 bytes</td>
<td>4096 - 65,536 bytes</td>
</tr>
<tr>
<td>Hit time</td>
<td>1 - 2 clock cycles</td>
<td>40 - 100 clock cycles</td>
</tr>
<tr>
<td>Miss Penalty</td>
<td>8 - 100 clock cycles</td>
<td>700,000 - 6,000,000 clock cycles</td>
</tr>
<tr>
<td>(Access time)</td>
<td>(6 - 60 clock cycles)</td>
<td>(500,000 - 4,000,000 clock cycles)</td>
</tr>
<tr>
<td>(Transfer time)</td>
<td>(2 - 40 clock cycles)</td>
<td>(200,000 - 2,000,000 clock cycles)</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>0.5 - 10%</td>
<td>0.00001 - 0.001%</td>
</tr>
<tr>
<td>Data memory size</td>
<td>0.016 - 1 MB</td>
<td>16 - 8192 MB</td>
</tr>
</tbody>
</table>

- VM has very high miss penalty
  - large pages (4 KB to MBs)
  - associative mapping of pages (typically fully associative)
  - software handling of misses (but not hits!!)
  - write-through not an option, only write-back
Translation Look-aside Buffer

- Address translation could be expensive to perform for every memory access
  - page tables are stored in main memory
  - need to access page table before accessing data location

- Solution is to remember the last address translation so the mapping lookup can be skipped
  - use a translation buffer to hold the last N translations
TLB: Making Address Translation Fast

Translation Lookaside Buffer: A cache for address translations
TLB and Cache

Virtual address
31 30 29 \ldots \cdot 15 14 13 12 11 10 9 \cdot \ldots 3 2 1 0

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

TLB

<table>
<thead>
<tr>
<th>Valid Dirty</th>
<th>Tag</th>
<th>Physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TLB hit

<table>
<thead>
<tr>
<th>Physical page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Physical address tag

<table>
<thead>
<tr>
<th>Byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache hit

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
</tbody>
</table>
Example: Page Table Size

- What is the total page table size if:
  - Virtual address is 32 bits
  - 8 Kbytes page size

- Assume that each page table entry is 4 bytes
- Size computation
Example: Address translation

- Using the page table shown, translate following 32-bit virtual addresses into physical addresses. Make entries in the TLB assuming LRU replacement.
  - The page size is 4 Kbytes
  - Addresses: 0x0000 3040, 0x0000 1040, 0x0000 2040

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 3040</td>
<td>0x0000 3040</td>
</tr>
<tr>
<td>0x0000 1040</td>
<td>0x0000 1040</td>
</tr>
<tr>
<td>0x0000 2040</td>
<td>0x0000 2040</td>
</tr>
</tbody>
</table>

Valid Physical Page Number

Page Table Register

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0010 0</td>
</tr>
<tr>
<td>1</td>
<td>0x000D 2</td>
</tr>
<tr>
<td>0</td>
<td>0xdead 1</td>
</tr>
<tr>
<td>1</td>
<td>0x0023 9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

TLB

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Access Problems

- **TLB Miss**
  - Entry does not exist in the TLB
  - A mechanism must be provided to bring a page table entry into the TLB

- **Page fault**
  - The valid bit is not set for the page table entry

- **Cache miss**
  - Tag mismatch or valid bit not set
  - Cache line is brought in (depending on the policy for a write)
Processing a Page Fault

- If the valid bit for a virtual page is off, page fault occurs
- CPU generates an exception
- OS takes control
- OS finds the page in the next level of hierarchy (disk)
- OS decides where to place the requested page in memory
- OS copies the page from next level of hierarchy to memory
- OS returns from the exception
- Program re-executes the same instruction
- Page translation finds valid bit on for the virtual page
- Data access succeeds
What is a Process?

- Program state consists of:
  - Page tables, PC and the registers
- This state is referred to as a process
- Process is an instance of a program executing on a CPU
Implementing Protection with VM

- Protection is essential for:
  - Allowing to share a single main memory among multiple processes
  - Prevent once process from writing into the memory space of another
  - Prevent a user process from modifying its own page tables
  - Controlling raw access to peripheral devices

- Hardware capabilities needed for protection
  - Two operating mode: user mode and kernel mode of execution
  - A portion of the CPU state that a user process can read, but not write
    - This is the usr/kernel mode bit
  - A mechanism to switch between user mode and kernel mode
    - Accomplished by a system call
Additional bits in the Page Table

• User or Kernel bit
  – This bit restricts access to some pages to kernel only

• Write bit
  – This bit restricts read-only or read/write access to a page

• Referenced bit
  – OS periodically sets this bit to zero
  – It is set by CPU hardware when the page is referenced
  – Used by OS for replacing the page with other memory pages

• Dirty bit
  – If a process writes to a page, the dirty bit is set
  – It is used by OS to write the page to secondary storage before replacing it
Virtual Memory Key Points

- How does virtual memory provide:
  - illusion of large main memory?
  - sharing?
  - performance?
  - protection?

- Virtual Memory requires twice as many memory accesses, so we cache page table entries in the TLB.

- Three things can go wrong on a memory access: cache miss, TLB miss, page fault.
## Example

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>C11</th>
<th>C12</th>
<th>C13</th>
<th>C14</th>
<th>C15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>ADD</strong> R1, R2, R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SW</strong> R1, 1000(R2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>LW</strong> R7, 2000(R2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>ADD</strong> R5, R7, R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>LW</strong> R8, 2004(R2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SW</strong> R7, 2008(R8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>ADD</strong> R8, R8, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>LW</strong> R9, 1012(R8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SW</strong> R9, 1016(R8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
M⇒M Forwarding for LW⇒SW
(Exercise 6.20 in the Textbook)
Example: Cache Organization

If you knew that the address 0x99B1 mapped to set 77 (base 10), what could you identify about the cache? You can assume that the cache is two-way set associative and that we are using 16-bit addresses and data.

1. How many sets are in the cache?
2. What is the block size?
3. What is the total size of the cache, including the valid bit and the tag bits?
4. What is the block offset for the address specified above?
5. What is the byte offset for the address specified above?
6. What is the tag for the address specified above?
Example: Cache Hit/Miss

Consider a two-way set associative cache with 4 sets. Assume 1 word (4 byte) blocks and LRU replacement scheme. Also assume that all valid bits are zero in the cache. For the following byte address trace presented below:

0, 4, 8, 11, 12, 20, 8, 33, 15, 27, 29, 50, 33, 10, 21, 2

1. Update the entries in the cache and show the final state of the cache, including valid and tag bits. In the data portion, show which bytes are stored.

2. How many hits are there in the cache?

3. How many misses are there in the cache?

4. If you were allowed to double the size of the cache, what change would you make to decrease the number of misses the most?
Example: Virtual Memory and TLB

A 32-bit processor has a virtual memory with page size of 4 Kbytes. The physical memory space is 256 Mbytes. There is a 4 entry fully associative TLB with LRU replacement policy. All the entries in the TLB are initially invalid. Following four accesses show virtual addresses and corresponding physical addresses.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 210a</td>
<td>0x000 010a</td>
</tr>
<tr>
<td>0x0000 0004</td>
<td>0x000 4004</td>
</tr>
<tr>
<td>0x0001 0010</td>
<td>0x000 1010</td>
</tr>
<tr>
<td>0x0030 2244</td>
<td>0x000 2244</td>
</tr>
</tbody>
</table>

1. In the TLB, how many bits are required for the tag?
2. In the TLB, how many physical page address bits are required?
3. Show the entries in the TLB after the access to the above four addresses are complete.
4. Explain why a "write bit" is needed in a TLB as well as in the page tables.