EN164: Design of Computing Systems
Topic 07: Memory System

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2012
Introduction

• Computer performance depends on:
  – Processor performance
  – Memory system performance
Introduction

- Up until now, assumed memory could be accessed in 1 clock cycle
Memory hierarchy

<table>
<thead>
<tr>
<th>Technology</th>
<th>cost / GB</th>
<th>Access time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>~ $10,000</td>
<td>~ 1 ns</td>
</tr>
<tr>
<td>DRAM</td>
<td>~ $100</td>
<td>~ 100 ns</td>
</tr>
<tr>
<td>magnetic hard disk</td>
<td>~ $1</td>
<td>~ 10,000,000 ns</td>
</tr>
</tbody>
</table>

Ideal memory
- Access time of SRAM
- Capacity and cost/GB of disk

2006 prices
Principle of locality

• Exploit locality to make memory accesses fast

• Temporal Locality:
  – Locality in time (e.g., if looked at a Web page recently, likely to look at it again soon)
  – If data used recently, likely to use it again soon
  – **How to exploit:** keep recently accessed data in higher levels of memory hierarchy

• Spatial Locality:
  – Locality in space (e.g., if read one page of book recently, likely to read nearby pages soon)
  – If data used recently, likely to use nearby data soon
  – **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too
Memory hierarchy levels

- **Block (aka line):** unit of copying
  - May be multiple words

- **If accessed data is present in upper level**
  - **Hit:** access satisfied by upper level
    - Hit ratio: hits/accesses

- **If accessed data is absent**
  - **Miss:** block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level
Example

- A program has 2,000 load and store instructions
- 1,250 of these data values found in cache
- The rest are supplied by other levels of memory hierarchy
- What are the hit and miss rates for the cache?

Hit Rate = 1250/2000 = 0.625
Miss Rate = 750/2000 = 0.375 = 1 – Hit Rate
Example

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$ cycle, $t_{\text{MM}} = 100$ cycles
- What is the average memory access time (AMAT) of the program from Example 1?

\[
\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}}(t_{\text{MM}}) \\
= [1 + 0.375(100)] \text{ cycles} \\
= 38.5 \text{ cycles}
\]
Cache memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
  - Fast (typically ~ 1 cycle access time)
  - Made out of SRAM cells
- Given accesses $X_1, \ldots, X_{n-1}, X_n$

<table>
<thead>
<tr>
<th>$X_4$</th>
<th>$X_{n-2}$</th>
<th>$X_{n-1}$</th>
<th>$X_2$</th>
<th>$X_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$X_4$</th>
<th>$X_{n-2}$</th>
<th>$X_{n-1}$</th>
<th>$X_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Before the reference to $X_n$
b. After the reference to $X_n$

- What data is held in the cache?
- How is data found?
- What data is replaced?
Direct mapped cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

- #Blocks is a power of 2
- Use low-order address bits
Example

<table>
<thead>
<tr>
<th>Address</th>
<th>2(^30) Word Main Memory</th>
<th>2(^3) Word Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>11...11111100</td>
<td>mem[0xFF...FC]</td>
<td></td>
</tr>
<tr>
<td>11...11111000</td>
<td>mem[0xFF...F8]</td>
<td></td>
</tr>
<tr>
<td>11...11110100</td>
<td>mem[0xFF...F4]</td>
<td></td>
</tr>
<tr>
<td>11...11110100</td>
<td>mem[0xFF...F0]</td>
<td></td>
</tr>
<tr>
<td>11...11110100</td>
<td>mem[0xFF...EC]</td>
<td></td>
</tr>
<tr>
<td>11...11101100</td>
<td>mem[0xFF...E8]</td>
<td></td>
</tr>
<tr>
<td>11...11101100</td>
<td>mem[0xFF...E4]</td>
<td></td>
</tr>
<tr>
<td>11...11101100</td>
<td>mem[0xFF...E0]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...D8]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...D4]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...D0]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...C8]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...C4]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...C0]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...B8]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...B4]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...B0]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...A8]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...A4]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...A0]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...98]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...94]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...90]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...88]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...84]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...80]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...78]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...74]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...70]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...68]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...64]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...60]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...58]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...54]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...50]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...48]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...44]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...40]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...38]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...34]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...30]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...28]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...24]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...20]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...1C]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...18]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...14]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...10]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...0C]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...08]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...04]</td>
<td></td>
</tr>
<tr>
<td>11...11100100</td>
<td>mem[0xFF...00]</td>
<td></td>
</tr>
</tbody>
</table>

Set Number

<table>
<thead>
<tr>
<th>Set Number</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
</tbody>
</table>
Tag and valid bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0
Cache hardware design

Memory Address

Tag Set Offset

27 3 00

V Tag Data

8-entry x (1+27+32)-bit SRAM

Hit

Data
Example

• How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?
Direct cache performance example

# MIPS assembly code

```
addi $t0, $0, 5
loop:  beq $t0, $0, done
   lw  $t1, 0x4($0)
   lw  $t2, 0xC($0)
   lw  $t3, 0x8($0)
addi $t0, $t0, -1
   j    loop

done:
```
Direct cache performance example

# MIPS assembly code

```
addi $t0, $0, 5
loop: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0xC($0)
lw $t3, 0x8($0)
addi $t0, $t0, -1
j loop
done:
```

Miss Rate = 3/15
= 20%

Temporal Locality
Compulsory Misses
Direct mapped cache: conflict

# MIPS assembly code

```
addi $t0, $0, 5
loop: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addi $t0, $t0, -1
j loop
done:
```
Direct cache map: conflict

# MIPS assembly code

```
addi $t0, $0, 5
loop:  beq  $t0, $0, done
lw    $t1, 0x4($0)
lw    $t2, 0x24($0)
addi $t0, $t0, -1
j     loop
done:
```

Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem[0x00...04]</td>
<td>00...00</td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td>mem[0x00...24]</td>
<td>00...00</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>

Miss Rate = 10/10
= 100%

Conflict Misses
Impact of larger block sizes

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
  - Block address = \[\lfloor 1200/16 \rfloor = 75\]
  - Block number = 75 modulo 64 = 11
Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks $\Rightarrow$ fewer of them
    - More competition $\Rightarrow$ increased miss rate

![Graph showing block size vs. miss rate](image-url)
Fully associative cache

- No conflict misses
- Expensive to build
N-way set associate cache
N-way set associative cache example

# MIPS assembly code

addi $t0, $0, 5
loop:  beq  $t0, $0, done
lw   $t1, 0x4($0)
lw   $t2, 0x24($0)
addi $t0, $t0, -1
j    loop
done:
N-way set associative cache example

# MIPS assembly code

```
addi $t0, $0, 5
loop:  beq $t0, $0, done
lw   $t1, 0x4($0)
lw   $t2, 0x24($0)
addi $t0, $t0, -1
j    loop
done:
```

Miss Rate = 2/10
= 20%

Associativity reduces conflict misses

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3
Set 2
Set 1
Set 0
4-way associative
Spectrum of associativity

- For a cache with 8 entries

One-way set 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>
</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>
<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>
<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>
Comparison example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
</tbody>
</table>
Comparison example

- **2-way set associative**

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[6]</td>
</tr>
</tbody>
</table>

- **Fully associative**

<table>
<thead>
<tr>
<th>Block address</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>miss</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[8]</td>
</tr>
</tbody>
</table>
How much associativity?

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
Handling cache writes

• *How to handle cache writes?*
  – Write through
  – Write back

• *How to handle write misses?*
Write through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full
Write back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first
Write allocation on write misses

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block
Sources of misses

- **Compulsory misses** (aka cold start misses)
  - First access to a block

- **Capacity misses**
  - Due to finite cache size
  - A replaced block is later accessed again

- **Conflict misses** (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size
## Cache design trade-offs

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effect on miss rate</th>
<th>Negative performance effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase cache size</td>
<td>Decrease capacity misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase associativity</td>
<td>Decrease conflict misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase block size</td>
<td>Decrease compulsory misses</td>
<td>Increases miss penalty.</td>
</tr>
</tbody>
</table>
Replacement choices

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- *What is the optimal entry to remove?*
- Random
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
LRU Replacement example

```mips
# MIPS assembly
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

(a)

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</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>
</tr>
</tbody>
</table>

Set Number
3 (11)
2 (10)
1 (01)
0 (00)

(b)

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</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>
</tr>
</tbody>
</table>

Set Number
3 (11)
2 (10)
1 (01)
0 (00)
LRU Replacement example

# MIPS assembly

```mips
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...000</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 1

Way 0

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

(a)

(b)
Multi-level caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache
Multi-level cache example

- **Given**
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns

- **With just primary cache**
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9
Multi-level cache example

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%

- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles

- Primary miss with L-2 miss
  - Extra penalty = 500 cycles

- CPI = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4

- Performance ratio = 9/3.4 = 2.6
Interactions between cache and superscalar architecture

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation
Cache summary

• Cache is on-chip memory built using 6T SRAM cells
• Cache organization:
  – Direct
  – Fully associative
  – N-way set associative
• Cache write techniques
• Cache block replacement techniques (for associative)
• Multi-level cache
DRAM is usually a shared resource among multiple processors, GPU and I/O devices -> a controller is needed to coordinate the access
DRAM organization (1-bit array)

- Reads are destructive; content must be restored after reading
- Capacitors are leaky so they must be periodically refreshed
- DRAM controller must each refresh each row (by reading and re-writing it) every 64 ms or less
- Refreshing contributes to the slow access of DRAMs
Multiple bit DRAM organization

In a x4 DRAM part, four arrays each read 1 data bit in unison, and the part sends out 4 bits of data each time the memory controller makes a column read request.
DRAM operation: bus transmission
DRAM operation: row access
DRAM operation: column access
DRAM operation: data transfer
DRAM operation: bus transmission
Simple asynchronous memory reading

Lower pin count (cost) by using same pins for row and column addresses
Synchronous DRAM (SDRAM) read

- To increase throughput, SDRAM outputs the data of sequential columns of the same row into the data bus (BURST mode). Now needs a clock.
- Matches well with cache blocks with multiple bytes.
Double Data Rate (DDR) SDRAM

Transfer data on positive and negative edges of the clock
DRAM memory bus organization

![Diagram of DRAM memory bus organization](image_url)

Word address  | Bank 0 | Word address  | Bank 1 | Word address  | Bank 2 | Word address  | Bank 3
---|---|---|---|---|---|---|
0 | | 1 | | 2 | | 3 |
4 | | 5 | | 6 | | 7 |
8 | | 9 | | 10 | | 11 |
12 | | 13 | | 14 | | 15 |
Memory interleaving

- **4-word wide memory**
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle

- **4-bank interleaved memory**
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle
DRAM connections

In PC

In smart phones/ tablets

DRAM die

Processor die

A PoP implementation

ADR connections

57
Virtual memory

• Each program uses *virtual addresses*
  – Entire virtual address space stored on a hard disk.
  – Subset of virtual address data in DRAM
  – CPU translates virtual addresses into *physical addresses*
  – Data not in DRAM is fetched from the hard disk

• Each program has its own virtual to physical mapping
  – Two programs can use the same virtual address for different data
  – Programs don’t need to be aware that others are running
  – One program (or virus) can’t corrupt the memory used by another
  – This is called *memory protection*
Virtual and physical addresses

Most accesses hit in physical memory
But programs have the large capacity of virtual memory
Virtual memory definitions

- **Page size**: amount of memory transferred from hard disk to DRAM at once
- **Address translation**: determining the physical address from the virtual address
- **Page table**: lookup table used to translate virtual addresses to physical addresses
Address translation

Virtual Address

<table>
<thead>
<tr>
<th>30</th>
<th>29</th>
<th>28</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>...</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>Page Offset</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>

Translation

<table>
<thead>
<tr>
<th>19</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>15</th>
</tr>
</thead>
</table>

PPN

<table>
<thead>
<tr>
<th>26</th>
<th>25</th>
<th>24</th>
<th>...</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>...</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Page Offset

Physical Address

© 2007 Elsevier, Inc. All rights reserved
Virtual memory example

**System:**
- Virtual memory size: 2 GB = $2^{31}$ bytes
- Physical memory size: 128 MB = $2^{27}$ bytes
- Page size: 4 KB = $2^{12}$ bytes

**Organization:**
- Virtual address: 31 bits
- Physical address: 27 bits
- Page offset: 12 bits
- # Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = 19 bits)
- # Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = 15 bits)
Virtual memory example

What is the physical address of virtual address \textit{0x247C}? 

- VPN = \textit{0x2}
- VPN \textit{0x2} maps to PPN \textit{0x7FFF}
- The lower 12 bits (page offset) is the same for virtual and physical addresses (\textit{0x47C})
- Physical address = \textit{0x7FFF47C}

19-bit virtual page numbers - 15-bit physical page numbers

© 2007 Elsevier, Inc. All rights reserved
Using page tables to translate addresses

• Page Table stores placement information
  – Array of page table entries, indexed by virtual page number
  – Page table register in CPU points to page table in physical memory
• If page is present in memory
  – PTE stores the physical page number
  – Plus other status bits (referenced, dirty, …)
• If page is not present
  – PTE can refer to location in swap space on disk
Translation using page table

Page table register

Virtual address

31 30 29 28 27 15 14 13 12 11 10 9 8 3 2 1 0

Virtual page number  Page offset

Valid

Physical page number

If 0 then page is not present in memory

Physical page number  Page offset

Physical address
Mapping pages to storage
What is the physical address of virtual address 0x5F20?

- VPN = 5
- Entry 5 in page table indicates VPN 5 is in physical page 1
- Physical address is 0x1F20
What is the physical address of virtual address 0x73E0?

- VPN = 7
- Entry 7 in page table is invalid, so the page is not in physical memory
- The virtual page must be *swapped* into physical memory from disk
Page table challenges

• Page table is large
  – usually located in physical memory

• Each load/store requires two main memory accesses:
  – one for translation (page table read)
  – one to access data (after translation)

• Cuts memory performance in half
  – Unless we get clever…
Last lecture summary
Replacement and writes

- **Replacement:** To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently

- **Writes:** Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical; use write-back
  - Dirty bit in PTE set when page is written
  - Write a physical page and reloading a different virtual page is called **swapping**
Fast translation using TLB

- Use a *translation lookaside buffer* (TLB)
  - Small cache of most recent translations
  - Reduces number of memory accesses required for *most* loads/stores from two to one

- TLB
  - Small: accessed in < 1 cycle
  - Typically 16 - 512 entries
  - Fully associative
  - > 99 % hit rates typical
  - Reduces # of memory accesses for most loads and stores from 2 to 1
Example: Two-entry TLB
TLB misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler

- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction
Memory protection

- Multiple programs (*processes*) run at once
- Each process has its own page table – special register holds the starting address of the table
- Each process can use entire virtual address space without worrying about where other programs are
- A process can only access physical pages mapped in its page table – can’t overwrite memory from another process

**Modifications to TLB:**
- Invalidate the TLB every time there is a context switch
- Or augment TLB with process id
1. TLB/Cache: physical tag, physical index

- **Virtual Address**: 20 bits
- **Page Offset**: 12 bits
- **4 KB Page Size**
- **Translation**
- **TLB**: 20 bits
- **20 bits**
- **12 bits**
- **1 Cycle**
- **16 KB Cache Size**
- **4 Bytes / Block**
- **Direct Mapped**
- **4096 Blocks**
- **Hit**
- **Data**

- **1 Cycle**
2. TLB/Cache: virtual tag, virtual index

- TLB is only accessed in case of miss
- Complications due to aliasing: Different virtual addresses for shared physical address
3. TLB/Cache: physical tag, virtual index

- Virtual address: 20 bits
- Page offset: 12 bits
- 4 KB page size

Translation in TLB:
- 1 cycle

Cache:
- 4KB cache size
- 4 bytes / block
- Direct mapped
- 1024 blocks
3. TLB/Cache: physical tag, virtual index

- TLB can now be accessed in parallel with cache
- Constrain cache design: *fundamental* cache size (# sets * block size = page size).
- Only way to increase cache size is by increasing associativity
Virtual memory summary

- Virtual memory increases **capacity and essential for multitasking** OS
- A subset of virtual pages are located in physical memory
- A **page table** maps virtual pages to physical pages – this is called address translation
- A **TLB** speeds up address translation
- Using different page tables for different programs provides **memory protection**
Pentium Pro memory hierarchy

- **Address Size:** 32 bits
- **VM Page Size:** 4 KB, 4 MB
- **TLB organization:** separate I and D TLBs (I-TLB: 32 entries, D-TLB: 64 entries) 4-way set associative LRU approximated hardware handles miss
- **L1 Cache:** 8 KB, separate I, D 4-way set associative LRU 32 byte block write back
- **L2 Cache:** 256 or 512 KB
Memory subsystem summary

• Fast memories are small, large memories are slow
  – We really want fast, large memories
  – Caching gives this illusion

• Principle of locality
  – Programs use a small part of their memory space frequently

• Memory hierarchy
  – L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk

• Virtual memory