Orange Coast College
Business Division
Computer Science Department

CS 116- Computer Architecture

The Memory Hierarchy
Direct-Mapped Cache

- Each memory location maps to exactly one location in the cache
  - One-to-many mapping
- Part of the memory address is used as tag
- Several items at the lower level share locations in the upper level
Direct Mapped Cache

• For MIPS:
  - Cache size $2^{10} = 1024$
  - Index bits = 10
  - Tag = 32 – 10 – 2 = 20
  - tag from cache compared against upper portion of address
  - If match ⇒ Entry in cache corresponds to requested address
  - If valid bit = 1 ⇒ Request hits ⇒ word supplied to processor
  - If valid bit = 0 ⇒ Request misses ⇒ bring word into cache
Direct Mapped Cache

- **Exercise:**
  - How many bits are needed for 64 KB of data

- **Assumptions:**
  - 1-word block
  - 32-bit address

```
Valid  Tag  Data
1 bit + (32-14-2)= 16 + 32 => total 49 bits / entry
```
Exercise: Answer

- **Formulas:**
  - Total number of bits = Cache entry size * block size
  - Cache entry size = Data size + Tag size + Valid bit size
- Cache has 64KB of data ( = $2^{14}$ words)
- One-word blocks = $2^{14}$ blocks
- Total bits needed = $2^{14} \times 49 = 2^{10} \times 2^4 \times 49$
  = $2^{10} \times 16 \times 49 = 784 \times 2^{10}$
  = 784 K-Bits

<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>

1 bit + (32-14-2) = 16 + 32 => total 49 bits / entry
Handling Cache Misses

• The action taken for cache miss depends on whether the action is to access instruction or data
  – Data Access
    • Stall the processor until the memory responds with the data
  – Instruction access:
    • The contents of the instruction register are invalid
      – Get the instruction from the lower-level memory into the cache
      – Decrement PC contents by 4 to get the address of the current instruction, since the PC is already incremented at the first clock cycle
Handling Cache Misses

- Instruction Access Steps:
  1. Send the original PC value (PC-4) to the memory
  2. Instruct main memory to perform read & wait for the memory to complete its access
  3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address in the tag field, and turning the valid bit on
  4. Restart the instruction execution at the first step, which will re-fetch the instruction, this time finding it in the case
Cache Example: DEC Station 3100

- Uses 2 separate caches
  - Instruction cache
  - Data cache
  - 16 K Blocks
  - 1 word per block
- Possible requests:
  - Read
  - Write
Cache Example: DEC Station 3100

- **Read Requests (e.g. lw)**
  - Send address to appropriate cache (Address comes either from PC or ALU)
  - If a hit: Requested word is available on data lines
  - If a miss:
    - Send address to main memory
    - When memory returns with the data, write it into cache

- **Write Requests (e.g. sw)**
  - After write, cache & memory might be inconsistent (i.e. have different values)
  - Possible approaches
    - Write-through
    - Write-back
Write-Through Approach

- **Idea:**
  - Always write data into both memory & cache
  - Or just write to cache, updating both tag & data
  - Or use cache buffer then write to memory

- **Steps:**
  1. Index the cache (Bits 15-2 of address)
  2. Write the tag, data, & valid bit into cache
  3. Write the word to main memory using the entire address
Write-through

- Advantages
  - Simple algorithm

- Disadvantages
  - Poor performance
  - If buffer is full, CPU stalls
    - Writing into main memory slows down the machine
Write-back Approach

- New values written only into cache
  - Modified block is written to lower level when replaced

- Steps:
  1. Index the cache
  2. Write the tag, data, & valid bit into cache
  3. The modified block is written to the memory only when it is replaced

- Advantages:
  - Improves performance

- Disadvantages:
  - Complex algorithm
Multiword Cache Blocks

- Taking advantage of spatial locality
  - Have a cache block larger than one word
  - An extra block offset field is used to control the MUX to select the right word in the block
  - Same mapping can be used to find a block in the cache

- Caches with multiword blocks use memory more efficiently
Multiword Cache Blocks

Address (Showing bit position)

 Valid bit
Hit
Tag
Index

Byte offset
Block offset

Data

16 bits Tag
128 bits Data

4K entries

16 bits Tag

16

32

32

32

32

MUX

32

16

32

32

32

16

32

32

32

16
Multiword Cache Blocks: Exercise

• Given:
  – Cache with 64 blocks
  – Block size = 16-bytes

• Required:
  – What block number does address 1200 map to?

• Answer:
  Block address = byte address / bytes per block
  Block # = (Block address) mod (# cache blocks)
  = (Byte address/Bytes per block) mod (# cache blocks)
  = (1200 / 16) mod (64)
  = 75 mod 64 = 11
Handling Misses for Multiword Blocks

• Read hits
  – This is what we want, proceed

• Read misses
  – Stall the CPU
  – Fetch the entire block from memory
  – Deliver block to cache
  – Restart
Handling Misses for Multiword Blocks

- **Write hits**
  - We can’t just write the tag & data
    - Have to check if the other words are from the same block in memory
  - What shall we do
    - First write the data & set the tag of the word
    - Compare the tags of all the words in that block
    - If tags are equal, proceed
    - If tags are not equal we have a write miss

- **Write misses**
  - Read the entire block into the cache, then write the word
Improving cache performance

- Reducing the miss rate
  - Different method for storing words in cache

- Reducing the miss penalty
  - Different way of organizing cache
Reduce miss rate

- Allow cache to place blocks more flexibly
  - Have discussed direct-mapped cache
    - A memory address can be located in only one place

- Two other ways to do it
  - Allow cache to place a block anywhere
    - Called fully-associative
    - Memory location can be associated with any block in cache
  - Or only allow a fixed number of locations where a block may be placed
    - Called set-associative
Fully-Associative Cache

• A cache structure in which a block can be replaced in any location in the memory
• Address is used as a tag
• If address matches tag, its associated data is accessed
• Tags must be compared simultaneously (associatively) with requested address
  – Can be very expensive for large caches
Set Associative

- A cache structure in which a block can be replaced in one of \( n \) blocks in the cache
  - Called \( n \)-way associative cache

- Consists of a number of sets that contain \( n \) blocks each
  - A memory block may be found in any of the sets.
  - Must search all sets simultaneously
Replacing a block

- Given that a block may be placed in \( n \)-locations
  - How do we decide where to place it?
    - LRU (Least Recently Used)
    - Random

- LRU
  - For each cache location, track last use of the block
    - Can be very costly for large associativity

- Random
  - Select a block at random and put the data there

- Performance
  - For large cache, performance difference is small
Reduce miss penalty

- The other method is to reduce the miss penalty
  - Insert another cache between memory and the processor
  - Called the L2 cache
- If data is in the L2 cache
  - Significant reduction of the miss penalty
- If not in L2 cache
  - Miss penalty increased
Memory System Supporting Cache

- Make reading multiple words easier by using banks of memory
- It can get a lot more complicated...

One-word-wide organization

Wide memory organization
Memory, buffer, & cache are wide

Interleaved organization
Narrow bus & cache, Interleaved memory
Memory System Supporting Cache

• One-Word-Wide
  – Memory is one word wide
  – All accesses are made sequentially

• Wide Memory
  – Increases the bandwidth by
    • widening the memory
    • Widening the buses between the processor & memory

• Interleaved
  – Increases the bandwidth by
    • Widening the memory
    • but not the interconnection bus
Performance

- Increasing the block size tends to decrease miss rate

- Use split caches because there is more spatial locality in code

### Table: Cache Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Block size in words</th>
<th>Instruction miss rate</th>
<th>Data miss rate</th>
<th>Effective combined miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1</td>
<td>6.1%</td>
<td>2.1%</td>
<td>5.4%</td>
</tr>
<tr>
<td>gcc</td>
<td>4</td>
<td>2.0%</td>
<td>1.7%</td>
<td>1.9%</td>
</tr>
<tr>
<td>spice</td>
<td>1</td>
<td>1.2%</td>
<td>1.3%</td>
<td>1.2%</td>
</tr>
<tr>
<td>spice</td>
<td>4</td>
<td>0.3%</td>
<td>0.6%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

### Graph: Miss Rate vs. Block Size

- Miss rate decreases as block size increases.
- Different line styles represent different cache sizes (1 KB, 8 KB, 16 KB, 64 KB, 256 KB).
- The trend is consistent across both instruction and data miss rates.
Virtual Memory

• Similar to caching, but between main memory & secondary storage instead of RAM & CPU
• Main memory acts as a “cache” for the secondary storage
• Only a portion of the program can be active in main memory, the rest will be in secondary storage
• Multiple program can share the same memory
• Programs sharing the memory change dynamically
Virtual Memory

- **Advantages**
  - Allows efficient & safe sharing of memory among multiple programs
  - Larger address space (Illusion of having more physical memory) without extra programming requirement

- **Disadvantages**
  - Overhead for
    - memory management
    - handling page faults
    - program relocation (converting virtual address space into physical address)
  - Programs need to be protected from each other
Virtual Memory

Virtual addresses

Address translation

Physical addresses

Disk addresses
Virtual Memory

• Virtual memory:
  – Intended to relieve the programmer from managing overlays
  – Virtual address space is broken into a number of equal-sized pages
  – Physical address space is also broken into the same page size (page frame)
  – Two or more pages can occupy the same page in physical address space
  – Consecutive pages in virtual address space need not be consecutive in physical address space
Virtual Memory

- Page
  - VM block
  - Usual size
    - 512 Bytes - 64 KBytes
    - Modern computer size up to 4 MBytes

- Page fault
  - Page is not in memory (~VM miss)

- Memory mapping
  - Address translation from virtual to physical address
Address Translation

• Example:
  – Page size = $2^{12} = 4$ KB
    • Dictates the maximum page offset
  – Number of physical pages allowed = $2^{18}$
  – Physical memory size = $2^{12} \times 2^{18} = 2^{30} = 1$ GB
  – Virtual address space = $2^{32} = 4$ GB
  – Virtual Address = Virtual page number + page offset
  – Physical Address = Physical page # + page offset
  – => Need to translate Virtual to Physical page number
    • Translation is performed using tables
    • Page offset stays the same for both virtual & physical addresses
Address Translation: Example

- Translating Virtual to Physical page number:

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

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

<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>11 10 9 8</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Pages Faults

• Data is not in memory and needs to be retrieved from disk

• Huge miss penalty
  – High access time
  – Pages should be fairly large (32-64 KB for new systems)

• Reducing page faults is important

• Faults can be handled in software instead of hardware
  – Have clever algorithms to choose how to place pages
Pages Faults: Replacement strategies

- Least-recently used (LRU):
  - Block replaced has been unused for the longest time
- First -in-first-out (FIFO)
  - Block replaced was in the memory the longest time
- Methods of replacement
  - Write-through
    - Too expensive
    - Write takes too long
  - Use write back
    - Only if the page is removed from main memory, write it back
Page Placement

• Fully associative page replacement
  – Placing the page
  – Finding a page again
Page Placement

• The operating system uses sophisticated algorithms & complex data structures to track page usage

• Goal:
  – Reduces page fault rate

• Method:
  – Locate page by using a page table
Page Table (PT)

- Maps virtual addresses into physical addresses
  - Resides in memory
  - One per program
- Not a cache => No tag field required
  - Contains physical page numbers of the program
- May contain entries for pages not in memory
- Indexed with page number
  - An index for each virtual page
- Each program has its own page table
Page Table (PT)

- Valid bit:
  - Used the same way it is used in cache
  - Valid bit = 0
    - Page not present in memory
    - If page is addressed, page fault occurs
  - Valid bit = 1
    - Entry contains physical page number

- Page table register:
  - Used to point to the page table
  - For each program there is a separate page table
  - On a context switch, address of page table stored into register
Page Table

Virtual page number

<table>
<thead>
<tr>
<th>Valid</th>
<th>Page table</th>
<th>Physical page or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical memory

Disk storage
Page Table (PT)

- Example:
  - Virtual address space = $2^{32}$
  - Physical memory size = $2^{30}$
  - Page size = $2^{12}$
  - Number of page table entries = $2^{32} - 2^{12} = 2^{20}$
  - Number of addressable pages in physical memory = $2^{30} - 2^{12} = 2^{18}$
  - # bits in each table entry = $18$ (page#) + (1 valid bit) = 19-bits wide

Diagram:

- Page table register
- Virtual address
- Virtual page number
- Page offset
- Valid
- Physical page number
- Page table (Size = 4KB)
- If 0 then page is not present in memory
- Physical page number
- Page offset
- Physical address
Page Table

- Problems:
  - Every access requires two steps:
    - Memory access to obtain physical address
    - If necessary handle page fault
    - Memory access to obtain data
  - PT can be very large
  - Several PTs needed for several programs
  - All must reside in memory
  - Too much memory for PTs, where do we put the programs?
  - Need to minimize memory space used by PTs
Page Table Performance

- To improve size, use one of the following
  - Start with a small PT, increase when necessary
  - Use the idea used for stack and heap:
    - Put 2 PTs in one area and let them grow in opposite directions
  - Apply hashing function to virtual address
    - Lookup will be more complicated
  - Use multiple-level PTs
    - PT should be paged in the same manner as virtual memory
Page Table: Solutions

• To improve time performance, use locality principle
  – Translation of a virtual page number will most probably be needed again
  – Use a cache to keep track of recently-used translation
    • Translation look-aside buffer
Translation Look-Aside Buffer (TLB)

- Cache that tracks recently used address mappings
  - Used to avoid an access to the page table
  - Contains a subset of virtual-to-physical page mappings that are in the Page Table
- Because TLB is a cache, it must have a tag field
- When no matching entry in TLB, Page Table examined
  - Supplies physical page number for the page
    - Used in building the TLB
    - Indicate that the page resides on disk,
      - Page fault occurs
Translation Look-aside Buffer

Virtual page #

Valid  Tag  TLB  Physical page address

1  1  1  1
1  1  1  0
1  0  1  1
1  1  1  0
1  1  1  1

Page table

Valid  Physical page or Disk address

1  1  1
1  1  1
1  0  1
1  1  1
1  1  1
1  1  1

Physical memory

Disk storage
Translation Look-aside Buffer (TLB)
Translation Look-aside Buffer (TLB)

- Handling “Write” Flowchart

```
<table>
<thead>
<tr>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB access</td>
</tr>
<tr>
<td>TLB hit?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>TLB miss exception</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Write?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Write access bit on?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Cache miss stall</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Cache hit?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Write protection exception</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Write data into cache, update the tag, and put the data and the address into the write buffer</td>
</tr>
</tbody>
</table>
```

Deliver data to the CPU
## Possible Events in TLB, VM, & Cache

<table>
<thead>
<tr>
<th>Cache</th>
<th>TLB</th>
<th>VM</th>
<th>Possible? Under what circumstances</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>hit</td>
<td>hit</td>
<td>Possible. But PT never checked if TLB hits</td>
</tr>
<tr>
<td>hit</td>
<td>miss</td>
<td>hit</td>
<td>TLB misses, entry found in PT; after retry, data found in cache</td>
</tr>
<tr>
<td>miss</td>
<td>miss</td>
<td>hit</td>
<td>TLB misses, entry found in PT; after retry, data misses in cache</td>
</tr>
<tr>
<td>miss</td>
<td>miss</td>
<td>miss</td>
<td>(Worst-case) TLB misses, followed by page fault; after retry, data must miss in cache</td>
</tr>
<tr>
<td>miss</td>
<td>hit</td>
<td>miss</td>
<td>Impossible; can’t have a translation in TLB if page not present in memory</td>
</tr>
<tr>
<td>hit</td>
<td>hit</td>
<td>miss</td>
<td>Impossible; can’t have a translation in TLB if page not present in memory</td>
</tr>
<tr>
<td>hit</td>
<td>miss</td>
<td>miss</td>
<td>Impossible; data can’t be allowed in cache if the page is not present in memory</td>
</tr>
</tbody>
</table>
Conclusion

- Processor speeds continue to increase very fast
  - much faster than either DRAM or disk access times
- Design challenge:
  - Dealing with this growing differences
- Trends:
  - Synchronous SRAMs (provide a burst of data)
  - Redesign DRAM chips to provide higher bandwidth or processing
  - Restructure code to increase locality
  - Use pre-fetching (make cache visible to ISA)