CSE 141 – Computer Architecture
Fall 2003

Lectures 14
Advanced Topics and Memory Hierarchy

Pramod V. Argade

---

CSE141: Introduction to Computer Architecture

Web-page:  http://www-cse.ucsd.edu/classes/fa03/cse141

Reading Assignment: Sections 6.1 through 6.9

Homework:  6.4, 6.10, 6.11, 6.12, 6.13, 6.20, 6.23, 6.26, 6.28, 6.29, 6.30

Due Date: Tuesday, November 25th

Next Quiz: Tuesday, December 2nd

Topic: Caches

Final Exam: Tuesday, December 9, 2003, 8:00 - 11:00am.

Room: Price Center Theatre
### Schedule

<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Day</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sep. 25</td>
<td>Thursday</td>
<td>Introduction, Ch. 1</td>
</tr>
<tr>
<td>2</td>
<td>Sep. 30</td>
<td>Tuesday</td>
<td>Performance, Ch. 2</td>
</tr>
<tr>
<td>3</td>
<td>Oct. 2</td>
<td>Thursday</td>
<td>ISA, Ch. 3</td>
</tr>
<tr>
<td>4</td>
<td>Oct. 7</td>
<td>Tuesday</td>
<td>Arithmetic, Ch. 4</td>
</tr>
<tr>
<td>5</td>
<td>Oct. 9</td>
<td>Thursday</td>
<td>Arithmetic, Ch. 4, Continued</td>
</tr>
<tr>
<td>6</td>
<td>Oct. 14</td>
<td>Tuesday</td>
<td>Single cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>7</td>
<td>Oct. 16</td>
<td>Thursday</td>
<td>Single-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>8</td>
<td>Oct. 21</td>
<td>Tuesday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>9</td>
<td>Oct. 23</td>
<td>Thursday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>10</td>
<td>Oct. 28</td>
<td>Tuesday</td>
<td>Classes cancelled due to wildfires</td>
</tr>
<tr>
<td>11</td>
<td>Oct. 30</td>
<td>Thursday</td>
<td>Exceptions and Review for Midterm</td>
</tr>
<tr>
<td>12</td>
<td>Nov. 4</td>
<td>Tuesday</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>13</td>
<td>Nov. 6</td>
<td>Thursday</td>
<td>Pipelining, Ch. 6</td>
</tr>
<tr>
<td>No Class</td>
<td>Nov. 11</td>
<td>Tuesday</td>
<td>Veteran's Day Holiday</td>
</tr>
<tr>
<td>14</td>
<td>Nov. 13</td>
<td>Thursday</td>
<td>Data hazards, Ch. 6</td>
</tr>
<tr>
<td>15</td>
<td>Nov. 18</td>
<td>Tuesday</td>
<td>Control hazards, Ch. 6</td>
</tr>
<tr>
<td>16</td>
<td>Nov. 20</td>
<td>Thursday</td>
<td>Memory Hierarchy and Caches, Ch. 7</td>
</tr>
<tr>
<td>17</td>
<td>Nov. 25</td>
<td>Tuesday</td>
<td>Memory Hierarchy and Caches, Ch. 7</td>
</tr>
<tr>
<td>No Class</td>
<td>Nov. 27</td>
<td>Thursday</td>
<td>Thanksgiving Holiday</td>
</tr>
<tr>
<td>18</td>
<td>Dec. 2</td>
<td>Tuesday</td>
<td>Virtual Memory, Ch. 7</td>
</tr>
<tr>
<td>19</td>
<td>Dec. 4</td>
<td>Thursday</td>
<td>Course Review</td>
</tr>
<tr>
<td>Dec. 9</td>
<td>Tuesday</td>
<td>Final Exam</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced Techniques
Advanced Techniques

- Superpipelining (Increases clock frequency)
  - More pipeline stages
  - Operand forwarding becomes complicated
  - Branch penalty is high
    - Must use branch prediction scheme

- Superscalar (Decreases CPI)
  - Multiple pipelines executing in parallel
  - Each pipeline may be dedicated to a particular task (integer, float, mem)
  - Challenge is finding instructions in parallel

- Dynamic pipelining (Decreases CPI)
  - Execute instruction out-of-order to avoid pipeline hazards
  - Retire them in execution order

Superscalar Datapath
Superscalar Issues

- Multiple instructions have to be fetched and decoded
- Additional ports are needed in the register file
  - Total 4 read ports, 2 write ports in our example
- Hardware resources have to be replicated
  - e.g. ALU, data forwarding paths, control logic, …
- Problems
  - How to find multiple instructions to issue at run time
  - Dependency on load instruction cannot be for multiple clocks
    - Determined by the number of instructions issued in parallel
  - Compiler technology need to statically schedule instructions
    - Breaks binary compatibility

Dynamic Pipeline Scheduling

- Three major sections
  - Instruction fetch and issue
  - Execute units
    - Each unit has reservation station to hold operands and operations
    - Instructions held in the reservation station until ready to execute
  - Commit unit
    - Common approach is in-order completion
    - Must discard instructions as a result of a mis-predicted branch
- Very complex to design and verify
Dynamically Scheduled Pipeline

Exceptions
Exception Handling in the Pipeline

- Consider arithmetic overflow exception
  - add $2, $6, $5
- Extra hardware
  - Note: add is in EX stage
  - Flush instructions that follow add
    - In IF stage, assert IF.flush
    - In ID stage, use mux added for LW followed by R-type stall (OR in ID.flush)
    - In EX stage, use EX.flush signal
  - Transfer control to PC = 0x4000 0040
  - Save PC + 4 in EPC
  - Save exception cause in Cause Register

Datapath and Control for Exceptions
Exception Handling in a Pipeline

0x40 sub $11, $2, $4
0x44 and $12, $2, $5
0x48 or $13, $2, $6
0x4c add $1, $2, $1
0x50 slt $15, $6, $7
0x54 lw $16, 50($7)

Note: ALU overflow signal is input the control unit

Issues in Handling an Exception

- Five instructions are active in the pipeline
- Exceptions are detected in different stage of the pipeline
  - Undefined instruction is discovered in ID stage
  - Overflow is detected in EX stage
  - Kernel call (i.e. OS call) is detected in EX stage
- Multiple exceptions may occur
  - Earliest instruction is generated interrupted
- Precise exception
  - EPC saves PC of the instruction that caused exception
  - This is required for virtual memory
- Imprecise exception
  - EPC may not save PC of the instruction that caused exception
    - For ease of implementation
Final Datapath and Control

Memory Hierarchy
Memory Systems

Memory Hierarchy in Computer Systems

Speed 1 ns 10's ns 100's ns (10s ms)

Size (bytes): 100s ~ KBytes ~ M Byttes ~ G Byttes ~ Tera Byttes
Memory Hierarchy

Who Cares about Memory Hierarchy?

- Processor vs Memory Performance

1980: no cache in microprocessor;
1995 2-level cache
Memory Subsystem Challenge

- Conflicting goals to provide:
  - Largest possible memory
  - At fastest access time
  - With lowest cost
- Processor speeds now exceed 3 Ghz (0.3 ns cycle time)
- DRAM access times are still ~10s of ns
- Serious Memory access gap
  - Every instruction has to be accessed from memory
  - ~15% of the instructions are load/store

Static RAM Cell and Data Access

<table>
<thead>
<tr>
<th>6-Transistor SRAM Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>word (row select)</td>
</tr>
<tr>
<td>bit</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- **Write:**
  1. Drive bit lines to data
  2. Select row
- **Read:**
  1. Precharge bit and bit to Vdd
  2. Select row
  3. Cell pulls one line low
  4. Sense amp on column detects difference between bit and bit

**Fast access, large area (6 transistors per cell)**
Dynamic RAM (DRAM) Cell and Data Access

- **Write:**
  - 1. Drive bit line to data
  - 2. Select row

- **Read:**
  - 1. Precharge bit line to Vdd
  - 2. Select row
  - 3. Cell and bit line share charges
    - Very small voltage changes on the bit line
  - 4. Sense voltage difference
    - Can detect changes of ~1 million electrons
  - 5. Write: restore the value

- **Refresh**
  - 1. Just do a dummy read to every cell

Slow access, small area (1 transistor per cell). Needs periodic refresh.

Magnetic Disk

Average access time =
Average seek time +
Average rotational delay +
Data transfer time +
Disk controller overhead

Slow access (~ ms), very large capacity (100’s GB)
Memory Locality

- Memory hierarchies take advantage of *memory locality*.
- *Memory locality* is the principle that future memory accesses are *near* past accesses.
- Memories take advantage of two types of locality
  - *Temporal locality* -- near in time
    - we will often access the same data again very soon
  - *Spatial locality* -- near in space/distance
    - our next access is often very close to our last access (or recent accesses).
- Consider following address sequence:
  1,2,3,4,5,6,7,8,8,47,8,9,8,10,8,8...

Locality and Caching

- A cache is a small amount of fast memory
- Memory hierarchies exploit locality by *caching* (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can’t build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity

SRAM (static RAM) -- 5-20 ns access time, very expensive
DRAM (dynamic RAM) -- 60-100 ns, cheaper
disk -- access time measured in milliseconds, very cheap
A Direct-mapped Cache

• How do determine whether a data item is in the cache?
• If a data item is in the cache, how do we find it?
  Cache location = (block address) modulo (Number of cache blocks in the cache)

Cache Fundamentals

• cache hit -- an access where the data is found in the cache.
• cache miss -- an access which isn’t
• hit time -- time to access the higher cache
• miss penalty -- time to move data from lower level to upper, then to cpu
• hit ratio -- percentage of time the data is found in the higher cache
• miss ratio -- (1 - hit ratio)
Cache Fundamentals, cont.

- *cache block size* or *cache line size* -- the amount of data that gets transferred on a cache miss.
- *instruction cache* -- cache that only holds instructions.
- *data cache* -- cache that only caches data.
- *unified cache* -- cache that holds both instructions and data.

A Direct-mapped Cache

<table>
<thead>
<tr>
<th>address string</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
<th>00001010</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
<td>00000101</td>
</tr>
<tr>
<td>10</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
</tr>
<tr>
<td>12</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>9</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
<td>00001001</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>21</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
<td>00001010</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
<td>00011000</td>
</tr>
<tr>
<td>14</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
<td>00001110</td>
</tr>
<tr>
<td>11</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
<td>00001011</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data in exactly one place is called a *direct-mapped cache*
A Fully-associative cache

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

<table>
<thead>
<tr>
<th>address string:</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 00010100</td>
</tr>
<tr>
<td>5   00000101</td>
</tr>
<tr>
<td>10  00001010</td>
</tr>
<tr>
<td>12  00001100</td>
</tr>
<tr>
<td>4   00000100</td>
</tr>
<tr>
<td>9   00001001</td>
</tr>
<tr>
<td>7   00000111</td>
</tr>
<tr>
<td>8   00001000</td>
</tr>
<tr>
<td>21  00010101</td>
</tr>
<tr>
<td>24  00011000</td>
</tr>
<tr>
<td>14  00001110</td>
</tr>
<tr>
<td>11  00001011</td>
</tr>
<tr>
<td>4   00000100</td>
</tr>
</tbody>
</table>

Valid bit indicates that entry is valid

The tag identifies the address of the cached data

<table>
<thead>
<tr>
<th>tag</th>
<th>v</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 entries, each block holds one word, any block can hold any word.

How is a Block found in the Cache?

- Address 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Valid Tag

Data

Index

Hit

Tag

0 1 2 3

... 1021 1022 1023

28 29 30

Pramod Argade

UCSD CSE 141, Fall 2003
Handling a Cache Miss

- **Read Miss:**
  - A mis-match on tag and/or Valid bit indicates a miss
  - Make read request to memory (via memory controller)
  - When memory returns the data write it into the cache
  - Return the data to the CPU

- **Write Miss:**
  - Write tag, valid bit and data into the memory
    - Works only if block size = word size
  - Should the data be written to memory also?

- **Write-through Cache**
  - Write data to cache as well as memory
  - Use write buffer so CPU can proceed with the following instructions

- **Write-back Cache**
  - Write cache data to memory when it is about to be overwritten for another address

Longer Cache Blocks

<table>
<thead>
<tr>
<th>address string:</th>
<th>00000101</th>
<th>00000110</th>
<th>00001100</th>
<th>00001000</th>
<th>00001001</th>
<th>00001110</th>
<th>00001011</th>
<th>00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>00000101</td>
<td>00000110</td>
<td>00001100</td>
<td>00001000</td>
<td>00001001</td>
<td>00001110</td>
<td>00001011</td>
<td>00000100</td>
</tr>
<tr>
<td>10</td>
<td>00001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>00000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>000000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>00000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>00001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>00010101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>00001010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>00001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00001011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Large cache blocks take advantage of *spatial locality*.
- Too large of a block size can waste cache space.
- Longer cache blocks require less tag space.
A 64 KB Cache using 16-byte Blocks

Summary

- Caches give an illusion of a large, cheap memory with the access time of a fast, expensive memory.

- Caches take advantage of memory locality, specifically temporal locality and spatial locality.