A computer system contains a hierarchy of storage devices with different costs, capacities, and access times. With a memory hierarchy, a faster storage device at one level of the hierarchy acts as a staging area for a slower storage device at the next lower level. Software that is well-written takes advantage of the hierarchy accessing the faster storage device at a particular level more frequently than the storage at the next level. As a programmer, understanding the memory hierarchy will result in better performance of applications.
Random Access Memory (RAM)

Features
- Basic storage unit is a cell (one bit per cell); RAM is traditionally packaged as a chip; multiple chips form memory
- Static RAM (SRAM)
  - Each cell implemented with a six-transistor circuit
  - Relatively insensitive to disturbances such as electrical noise, radiation, etc.
  - Faster and more expensive than DRAM
- Dynamic RAM (DRAM)
  - Each bit stored as charge on a capacitor
  - Value must be refreshed every 10-100 ms
  - Sensitive to disturbances
  - Slower and cheaper than SRAM

<table>
<thead>
<tr>
<th>Transistors per bit</th>
<th>Access time</th>
<th>Needs refresh?</th>
<th>Sensitive?</th>
<th>Cost</th>
<th>Power requirements</th>
<th>Power dissipation</th>
<th>Chip density</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>No</td>
<td>100x</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td>high</td>
<td>low</td>
</tr>
</tbody>
</table>

Conventional DRAM Organization
- \((d \times w)\) bit DRAM chip is organized as \(d\) supercells of size \(w\) bits
- Usually \(w=8\), i.e. one byte

Reading 16x8 DRAM Supercell [2,1]
Step 1: Row access strobe (RAS) selects row 2. Row 2 copied from DRAM array to row buffer.
Reading 16x8 DRAM Supercell [2,1] (cont.)

Step 2: Column access strobe (CAS) selects column 1. Supercell [2,1] copied from buffer to data lines to CPU.

- Step 3: Since a read is distractive, the internal row buffer has to be written back into the corresponding row of DRAM array after each read.

Conventional 4Gbit DRAM Organization

- 4Gbit DRAM can be organized as 512M supercells of size 8 bits

\[ 512M = 2^9 \times 2^{20} = 2^{14} \times 2^{15} = 16,384 \times 32,768 \]

Writing DRAM

- CPU provides address and data to be written
- Step 1 write is identical to step 1 read.
- Step 2 write includes updating the appropriate cell in the internal row buffer with data received from CPU
- Step 3 write is identical to step 3 read.

4GB Memory out of Eight 512Mx8 DRAMs
Enhanced DRAMs

- Enhanced DRAMS have optimizations that improve the speed with which the basic DRAM cells can be accessed.
- Examples:
  - Fast page mode DRAM (FPM DRAM); up to 1996
  - Extended data out DRAM (EDO DRAM); 1996-99
  - Synchronous DRAM (SDRAM)
  - Double Data-Rate Synchronous DRAM (DDR SDRAM)
  - Rambus DRAM (RDRAM)

Prices of Six Generations of DRAMs

<table>
<thead>
<tr>
<th>Year introduced</th>
<th>Chip size</th>
<th>$ per MB</th>
<th>Total access time in a 300 mm wafer</th>
<th>Average access across access time to minimum time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64 Kbit</td>
<td>$1.56/MB</td>
<td>290 ns</td>
<td>100 ns</td>
</tr>
<tr>
<td>1983</td>
<td>256 Kbit</td>
<td>$0.75/MB</td>
<td>185 ns</td>
<td>100 ns</td>
</tr>
<tr>
<td>1985</td>
<td>1 Mbit</td>
<td>$0.75/MB</td>
<td>135 ns</td>
<td>40 ns</td>
</tr>
<tr>
<td>1989</td>
<td>4 Mbit</td>
<td>$0.50/MB</td>
<td>110 ns</td>
<td>40 ns</td>
</tr>
<tr>
<td>1992</td>
<td>16 Mbit</td>
<td>$0.50/MB</td>
<td>80 ns</td>
<td>30 ns</td>
</tr>
<tr>
<td>1996</td>
<td>64 Mbit</td>
<td>$0.50/MB</td>
<td>60 ns</td>
<td>30 ns</td>
</tr>
<tr>
<td>1996</td>
<td>128 Mbit</td>
<td>$0.00/MB</td>
<td>60 ns</td>
<td>30 ns</td>
</tr>
<tr>
<td>2000</td>
<td>256 Mbit</td>
<td>$0.00/MB</td>
<td>50 ns</td>
<td>7 ns</td>
</tr>
<tr>
<td>2004</td>
<td>512 Mbit</td>
<td>$0.00/MB</td>
<td>50 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>2007</td>
<td>1 Gbit</td>
<td>$0.00/MB</td>
<td>40 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>2010</td>
<td>2 Gbit</td>
<td>$0.00/MB</td>
<td>30 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>2012</td>
<td>4 Gbit</td>
<td>$0.00/MB</td>
<td>30 ns</td>
<td>5 ns</td>
</tr>
</tbody>
</table>

- DRAM size increased by multiples of four approximately once every three years until 1996, and thereafter considerably slower.
- The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand.
- The cost per gigabyte ($10^9$ bytes) is not adjusted for inflation.
Nonvolatile Memory

- Information retained if supply voltage is turned off
- Referred to as read-only memories (ROM), although some may be written to as well as read
- Read-only memory (ROM): programmed during production
- Programmable ROM (PROM): fuse associated with cell that is blown once by zapping with current; can be programmed once
- Erasable PROM (EPROM): cells cleared by shining ultraviolet light, special device used to write 1's; can be erased and reprogrammed about 1000 times
- Electrically erasable PROM (EEPROM): similar to EPROM but does not require a physically separate programming device, can be re-programmed in place on printed circuit cards; can be reprogrammed about 100,000 times
- Flash Memory
  - Based on EEPROM technology

Solid State Disk - SSD

- SSD package plugs into a standard disk slot on the I/O bus (typically USB or SATA) and behaves like a disk reading and writing logical blocks.
- Consists of one or more flash memory chips and a flash translation layer (hardware/firmware device) that plays the same role as a disk controller.
- Advantages of SSD over rotating disks:
  - No moving parts – semiconductor memory is more rugged
  - Much faster random access times
  - Use less power
- Disadvantages of SSD over rotating disks:
  - SSDs wear out with usage
  - 10-20 times more expensive than disks

Basic (Single CPU) Computer Structure

- CPU and device controllers connect through common bus providing access to shared memory

Solid State Disk – SSD (cont.)

- Pages: 512B to 4KB, Blocks: 32 to 128 pages
- Data read in units of pages.
- Page can be written only after its block has been erased
- A block wears out after 100,000 repeated writes.
Moving-Head Disk Mechanism

- A sector (usually 512 bytes) is a basic unit of transfer (read/write)

SSD Characteristics

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential read throughput</td>
<td>250 MB/s</td>
</tr>
<tr>
<td>Random read throughput</td>
<td>140 MB/s</td>
</tr>
<tr>
<td>Sequential write throughput</td>
<td>170 MB/s</td>
</tr>
<tr>
<td>Random write throughput</td>
<td>14 MB/s</td>
</tr>
<tr>
<td>Random read access</td>
<td>30 us</td>
</tr>
<tr>
<td>Random write access</td>
<td>300 us</td>
</tr>
</tbody>
</table>

Growth in Microprocessor Performance

- Mismatch between CPU performance growth and memory performance growth
  - "memory wall"
- Importance of cache.

Growth in Performance of RAM & CPU
**CPU Trends: “Power Wall”**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8080</td>
<td>386</td>
<td>Pentium</td>
<td>P-4</td>
<td>P-4</td>
<td>Core 2</td>
<td>Core i7</td>
<td>—</td>
</tr>
<tr>
<td>Clock rate (MHz)</td>
<td>1</td>
<td>20</td>
<td>150</td>
<td>600</td>
<td>3300</td>
<td>2000</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>Cycle time (ns)</td>
<td>1000</td>
<td>50</td>
<td>6</td>
<td>1.6</td>
<td>0.3</td>
<td>0.50</td>
<td>0.4</td>
<td>2500</td>
</tr>
<tr>
<td>Cores</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Effective cycle time (ns)</td>
<td>1000</td>
<td>50</td>
<td>6</td>
<td>1.6</td>
<td>0.3</td>
<td>0.25</td>
<td>0.1</td>
<td>10,000</td>
</tr>
</tbody>
</table>

- At about the same time, besides “memory wall” and “power wall”, processor designers also reached limits in taking advantage of instruction level parallelism – ILP in (sequential) programs.
- Since early 2000’s processors have not been (significantly) getting faster.
- Instead, multi-core processors.

**Principle of Locality of Reference**

- Programs access a small proportion of their address space at any time and they tend to reuse instructions and data they have used recently.
  - temporal locality – recently accessed items are likely to be accessed soon,
  - spatial locality – items near those accessed recently are likely to be accessed soon, i.e. items near one another tend to be referenced close together in time.
- An implication of principle of locality is that we can predict with reasonable accuracy what instructions and data a program will use in near future based on its accesses in the recent past.
- Principle of locality applies more strongly to code accesses than data accesses.

**Taking Advantage of Locality**

- Use memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory (virtual memory)
- Copy recently accessed (and nearby) items from DRAM memory to smaller SRAM memory
  - Cache attached to CPU

**Typical System Organization Without Cache**

- Register file
- ALU
- System bus
- Memory bus
- Main memory
- CPU chip
- s/l bridge
- I/O bus
- Memory controller
- USB controller
- Graphics adapter
- Disk controller
- Monitor
- Mouse Keyboard
- Expansion slots for other devices such as network adapters.
**Typical Processor Organization with Cache**

Cache is fast but because of that it has to be small. Why?

**Model of Memory + Cache + CPU System**

**Basics of Cache Operation**

- We will first (and mostly) consider a cache read operation, since it is more important. Why?
- When CPU provides an address requesting a content of a given main memory location:
  - first check the cache for the content; caches include a tag in each cache entry to identify the memory address of a block,
  - if the block present, this is a hit, get the content (fast) and CPU proceeds normally, without (or small) delay,
  - if the block not present, this is a miss, stall CPU and read the block with the required content from main memory,
  - long CPU slowdown: *miss penalty* \(\rightarrow\) time to access main memory and to place a block into the cache,
  - And now (after miss penalty) CPU gets the required content.

**Tags and Valid Bits**

- How do we know what block is stored in a given cache location?
  - store block address as well as the data in a cache entry
  - actually, it may only need the high-order bits of block address called the tag
- What if there is no data in a location?
  - introduce valid bit in each entry
  - valid bit: 1 = present, 0 = not present
  - initially 0
**Cache Example**

- CPU generates 4-byte word read at address 100 (16 bit address assumed), i.e., read for contents of bytes at addresses 100-103
- Since cache is empty → cache miss

**Cache after 4B Read at Address 100**

1. Block of 16 byte read from RAM and stored in cache
2. Cache entry chosen according to cache placement algorithm (direct mapped cache assumed).

**Cache after 4B Read at Address 204**

1. Block of 16 byte read from RAM and stored in cache
2. Cache entry chosen according to cache placement algorithm (direct mapped cache assumed).

**Cache & DRAM Memory: Performance**

- \( t_2 \): main memory access time (e.g., 50 nsec)
- \( t_1 \): cache access time (e.g., 2 nsec)
- Hit ratio is a ratio of a number of hits and a total number of memory accesses; Miss ratio = 1 – Hit ratio

\[
\text{Average Access Time} = \text{Cache Access Time} + (1 - \text{Hit ratio}) \times \text{Miss Penalty}
\]

- Because of locality of reference, hit rates are normally well over 90%
Direct Mapped Cache

- In direct mapped caches, a block from memory has only one cache entry where it has to be stored.
- Location of cache entry is determined by block address and number of entries in the cache:
  - location = (block address) mod (number of entries in cache)
- Block address includes all address bits excluding block offset bits, i.e. rightmost $n$ bits where:
  - $2^n = n$ number of bytes in a block
- Example: 16 bit addresses and block size = 8 bytes
  - then address: $0x1234 = 0001 0010 0011 0100_2$ has block address $0x0246 = 0 0010 0100 0110$
  - since block offset = 3 ($2^3 = 8$)
- then if the cache has 16 entries → location is 0110, i.e. 4 rightmost bits of block address, since $2^4 = 16$, that is the index.

Direct Mapped Cache Example

- Assume processor with 10-bit addresses & a direct mapped cache with 8 entries, and 4 byte blocks;
- 4 byte blocks $= 2^2 = 2^{\text{block offset}} \rightarrow \text{block offset} = 2$
- block address $= 10-2 = 8$ bits
- 8 entries $= 2^3 = 2^{\text{index}} \rightarrow \text{index} = 3$ bits
- Address format = 5 bit (tag) + 3 bits (index) + 2 bits (block offset)

<table>
<thead>
<tr>
<th>V</th>
<th>Tag = 5 bits</th>
<th>Data = 32 bits = 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache Example: Access 1

<table>
<thead>
<tr>
<th>Word address</th>
<th>Binary address</th>
<th>Hit/miss</th>
<th>Cache entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>00010 110 00</td>
<td>Miss</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>00010</td>
<td>Mem[88-91]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache Example: Access 2

<table>
<thead>
<tr>
<th>Word address</th>
<th>Binary address</th>
<th>Hit/miss</th>
<th>Cache entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>00011 010 00</td>
<td>Miss</td>
<td>010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>00011</td>
<td>Mem[104-107]</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>00010</td>
<td>Mem[88-91]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cache Example: Accesses 3, 4

<table>
<thead>
<tr>
<th>Word address</th>
<th>Binary address</th>
<th>Hit/miss</th>
<th>Cache entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>00010 110 00</td>
<td>Hit</td>
<td>110</td>
</tr>
<tr>
<td>104</td>
<td>00011 010 00</td>
<td>Hit</td>
<td>010</td>
</tr>
</tbody>
</table>

Index V Tag Data

000 Y 00010 Mem[64-67]
001 N
010 Y 00000 Mem[12-15]
011 Y 00000 Mem[12-15]
100 N
101 N
110 Y 00010 Mem[88-91]
111 N

Cache Example: Accesses 5, 6, 7

<table>
<thead>
<tr>
<th>Word address</th>
<th>Binary address</th>
<th>Hit/miss</th>
<th>Cache entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>00010 000 00</td>
<td>Miss</td>
<td>000</td>
</tr>
<tr>
<td>12</td>
<td>00000 011 00</td>
<td>Miss</td>
<td>011</td>
</tr>
<tr>
<td>64</td>
<td>00010 000 00</td>
<td>Hit</td>
<td>000</td>
</tr>
</tbody>
</table>

Index V Tag Data

000 Y 00010 Mem[64-67]
001 N
010 Y 00011 Mem[104-107]
100 N
101 N
110 Y 00010 Mem[88-91]
111 N

Cache Example: Accesses 8

<table>
<thead>
<tr>
<th>Word address</th>
<th>Binary address</th>
<th>Hit/miss</th>
<th>Cache entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>00010 010 00</td>
<td>Miss</td>
<td>010</td>
</tr>
</tbody>
</table>

Index V Tag Data

000 Y 00010 Mem[64-67]
001 N
010 Y 00010 Mem[72-75]
011 Y 00000 Mem[12-15]
100 N
101 N
110 Y 00010 Mem[88-91]
111 N

Direct Mapped Cache

- Direct mapped caches: only one choice for cache entry
- Location of cache entry determined by block address
  - location = (block address) mod (number of entries in cache)
- Use low-order address bits as a location for cache entry
**Direct Mapping Cache: 1 × 4-byte Blocks**

- Block offset = 2 bits since $2^2 = 4$ bytes
- Here, block = data then block offset = byte offset
- Index = 14 bits since $2^{14} = 16K$ number of cache entries

**Cache after Addresses 100 and 204**

- address $100_{16} = 0000000000000000 \quad 0000000001101000\ 00$
  - block address = $0000000000000000 \quad 0000000001101001\ 2$
  - 4-byte block [100-103] will be stored in cache entry $25 = 0000000001101001\ 2$
  - Note: bytes with addresses 100-103 have same block address

- address $204_{16} = 0000000000000000 \quad 0000000011001100\ 00$
  - block address = $0000000000000000 \quad 0000000011001101\ 2$
  - 4-byte block [204-207] will be stored in cache entry $51 = 0000000011001101\ 2$
  - Note: bytes with addresses 204-207 have same block address

---

**Direct Mapping Cache: 4 × 4-byte Blocks**

- Block offset = 4 bits ($2^4 = 16$ bytes); Index = 12 bits ($2^{12} = 4K$ entries)
- Since 4-byte words (data) → block offset = 2 bits word offset + 2 bits byte offset since $4 = 2^2 = 2$ byte offset

**Cache after Addresses 100 and 204**

- address $100_{16} = 0000000000000000 \quad 0000000001101000\ 00$
  - block address = $0000000000000000 \quad 0000000001101001\ 2$
  - 16-byte block [96-111] will be stored in cache entry $6 = 0000000001101010\ 2$
  - Note: bytes with addresses 96-111 have same block address

- address $204_{16} = 0000000000000000 \quad 0000000011001100\ 00$
  - block address = $0000000000000000 \quad 0000000011001101\ 2$
  - 16-byte block [192-207] will be stored in cache entry $12 = 0000000011001110\ 2$
  - Note: bytes with addresses 192-207 have same block address
Block Size Consideration

- Cache with 4-byte blocks had 16K entries → total of 64KB data
- Cache with 16-byte blocks had 4K entries → total of 64KB data
- Thus, both caches can accommodate the same amount of data

<table>
<thead>
<tr>
<th>Program</th>
<th>Block size in bytes</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>4</td>
<td>6.1%</td>
<td>2.1%</td>
<td>5.4%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.0%</td>
<td>1.7%</td>
<td>1.9%</td>
</tr>
<tr>
<td>spice</td>
<td>4</td>
<td>1.2%</td>
<td>0.6%</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.3%</td>
<td>0.6%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

Larger blocks reduce miss rate due to spatial locality. But for a fixed-sized cache, larger blocks → fewer of them → more competition → may increased miss rate. Larger miss penalty may override benefit of reduced miss rate. Thus, keep in mind, the miss rate is not the only parameter:

\[
\text{AverageAccessTime} = \text{CacheAccessTime} + \text{MissRate} \times \text{MissPenalty}
\]

Main Memory Supporting Caches

- DRAM memory has a width of its read/write operations determined by a width of its bus data lines.
- Numbers used in examples that follow for reading from main memory:
  - 2nsec for address transfer from CPU to memory controller (mostly propagation delay),
  - 50nsec per DRAM access,
  - 2nsec per data transfer from memory controller to cache (mostly propagation delay)

4-Byte Main Memory Bus

- For 16-byte block, and 4-byte-wide DRAM bus:
  \[
  \text{Miss penalty} = 2 + 4 \times 50 + 4 \times 2 = 210 \text{nsec},
  \text{Bandwidth} = \frac{16 \text{ bytes}}{210} = 0.08 \text{ bytes/nsec}.
  \]

- Although caches are interested in low-latency memory, it is generally easier to improve memory bandwidth with new memory organization than it is to reduce memory latency.

16-Byte (Wider) Main Memory Bus

- For 16-byte block, and 16-byte-wide DRAM bus:
  \[
  \text{Miss penalty} = 2 + 50 + 2 = 54 \text{nsec},
  \text{Bandwidth} = \frac{16 \text{ bytes}}{54} = 0.30 \text{ bytes/nsec}.
  \]

- CPU accesses a word at a time, so a need for a multiplexer.
Wider Main Memory Bus & Level-2 Cache

- But the multiplexer is on the critical timing path.
- Level-2 cache helps since the multiplexing is now between level-1 and level-2 caches and not on critical timing path.

Interleaved Memory Organization

- This is four-way interleaved memory with 4 byte memory bus
- Miss penalty = 2 + 50 + 4 \times 2 = 60\text{nsec}
- Throughput = 16 \text{ bytes} / 60 = 0.27 \text{ bytes/nsec}

Multilevel Caches

- Level-1 (primary) cache attached to CPU
  - small, but very fast
- Level-2 (secondary) cache services misses from level-1 cache
  - larger, slower, but still faster than main memory
- Main memory services level-2 cache misses
- Some high-end systems include level-3 cache
- Level-1 cache: focus on minimal hit time
- Level-2 cache:
  - focus on low miss rate to avoid main memory access
  - hit time has less overall impact

Multilevel Caches (cont.)

- Increased logic density => on-chip cache with processor
  - internal cache: level 1
  - internal cache: level 2
  - external or internal cache: level 3
- Unified cache, i.e. a cache for both instruction and data
  - balances the load between instruction and data fetches,
  - only one cache needs to be designed and implemented;
- Split cache:
  - data cache (d-cache) and instruction cache (i-cache)
  - essential for pipelined & other advanced architectures;
- Level 1 caches are normally split caches.
Associative Caches

- In addition to direct mapped, two more cache organizations
  - Fully associative caches:
    - allow a given block to go in any cache entry
    - requires all entries to be searched at once
    - tag comparator per entry (expensive)
    - Note: no index, thus a tag equals a block address
  - n-way set associative caches:
    - each set contains n entries (blocks)
    - block number determines which set in the cache
      set number = (block number) mod (number of sets)
    - search only n entries in a given set (at once)
    - n comparators (less expensive)

Illustration of Cache Organizations

- Fully associative:
  - block 12 can go anywhere
- Direct mapped:
  - block 12 can go only into block 4
- 2 way set associative:
  - block 12 can go anywhere in set 0

4-Way Set Associate Cache with 4-byte Blocks

- There are 256 sets
- Indexing sets

Set Associative Cache: Example

- Consider 2-way set associative cache with 4 sets, and 8-byte blocks. Assume 16-bit address.
  a. provide address format.
    8-byte blocks → 3-bit block offset; 4 sets → 2-bit index
    4-byte data (words) → 2-bit byte offset
    3-bit block offset = 2-bit byte offset + 1-bit word offset
    16-bit address = 11-bit tag + 2-bit index+1-bit word offset +2-bit byte offset
  b. Provide cache layout
  c. For address sequence: 8, 0, 52, 20, 56, 16, 24, 116, 20, 8, 16
     indicate hit/miss and content of the cache, initially empty,
### Comparing Finding a Block

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Location method</th>
<th>Tag comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>Index entry</td>
<td>1</td>
</tr>
<tr>
<td>n-way set associative</td>
<td>Index set, then search n entries within the set</td>
<td>n</td>
</tr>
<tr>
<td>Fully associative</td>
<td>Search all entries</td>
<td>Number of entries</td>
</tr>
</tbody>
</table>

- Reduce a number of comparisons to reduce cost.

### Replacement Policy

- Refers to the situation when on a miss there is no room for a new block and one of existing blocks in the cache has to be removed from caches.
- Direct mapped cache: no choice.
- N-way set associative cache:
  - choose among \( n \) entries in the set;
- Fully associative cache:
  - choose among all entries in the cache;
- Least-recently used (LRU) replacement policy:
  - choose the one unused for the longest time,
  - simple for 2-way, manageable for up to 16-way, probably too inefficient beyond that;
- Random replacement policy also possibility;
Cache Writing on Hit

- Write is more complex and longer than read since:
  - write and tag comparison can not proceed simultaneously as read and tag comparison can in the case of cache read,
  - only a portion of the block has to be updated;
- Two approaches when cache hit: write through & write back
- Write through technique:
  - updates a main memory and the block in cache,
  - but it makes writes take longer and it does not take advantage of cache, although simple to implement.
- Improvements to this technique → special write buffer:
  - it holds data waiting to be written to memory,
  - CPU continues immediately, and CPU stalls on write only if write buffer is already full.

Cache Writing on Hit (cont.)

- Write back technique:
  - just update the block in cache,
  - keep track of whether or not a block has been updated; each entry has a dirty bit for that purpose.
  - update in memory when cache entry has to be replaced
- Cache and memory are inconsistent; problem in multi-processor (-core) systems. Why?
  - When a dirty block is to be replaced:
    - write it back to memory,
    - can also use write buffer,
- Write through is usually found in level 1 data caches backed by level 2 cache that uses write back.

Cache Writing on Miss

- What should happen on a write miss?
- Write-allocate: load block into cache, and update in cache (good if more writes to the location follow);
  - Write-back usually uses write-allocate;
- No-write-allocate: writes immediately only to main memory;
  - Write-through usually uses no-write-allocate

Intel Core i7 Cache Hierarchy

Processor package Intel Core i7 (4 cores):
- L1: i-cache and d-cache: 32 KB each, 8-way
- Access: 4 cycles
- L2: unified cache: 256 KB, 8-way
- Access: 13 cycles
- L3: unified cache:
  - 8 MB, 16-way
  - Access: 30-40 cycles
- Block size: 64 bytes for all caches.