The Memory Hierarchy

(Slides adapted from CSAPP)
Announcements

- Quiz2 grades are posted
- HW2 due: Fri. 11:59 pm
Summary of previous class

- Memory hierarchy overview
  - Huge difference between register based operations and memory based operations
  - Caches to increase memory access efficiency
    - Temporal and spatial Locality

- This class: More on memory hierarchy
  - Locality examples
  - Cache organization and addressing
Agenda

- Storage technologies and trends
  - DRAM
  - SRAM
  - DISC
  - Solid State Disks

- Locality of reference

- Caches
  - Direct memory mapping
  - 2-set associative
  - Full associative
Random-Access Memory (RAM)

- **Key features**
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a cell (one bit per cell).
  - Multiple RAM chips form a memory.

- **Static RAM (SRAM)**
  - Each cell stores a bit with a four or six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to electrical noise (EMI), radiation, etc.
  - Faster and more expensive than DRAM.

- **Dynamic RAM (DRAM)**
  - Each cell stores bit with a capacitor. One transistor is used for access.
  - Value must be refreshed every 10-100 ms.
  - More sensitive to disturbances (EMI, radiation,…) than SRAM.
  - Slower and cheaper than SRAM.
# SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th>Trans. per bit</th>
<th>Access time</th>
<th>Needs refresh?</th>
<th>Noise sensitive?</th>
<th>Cost</th>
<th>Applications</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>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>
Conventional DRAM Organization

- \( d \times w \) DRAM:
  - \( dw \) total bits organized as \( d \) supercells of size \( w \) bits

![Diagram of 16x8 DRAM chip with internal row buffer and memory controller](image-url)
Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2.
Step 1(b): Row 2 copied from DRAM array to row buffer.
Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.
Memory Modules

64 MB memory module consisting of eight 8Mx8 DRAMs

addr (row = i, col = j)

64-bit doubleword at main memory address A

Memory controller

64-bit doubleword
Enhanced DRAMs

- Basic DRAM cell has not changed since its invention in 1966.
  - Commercialized by Intel in 1970.

- DRAM cores with better interface logic and faster I/O:
  - Synchronous DRAM (SDRAM)
    - Uses a conventional clock signal instead of asynchronous control
    - Allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)

- Double data-rate synchronous DRAM (DDR SDRAM)
  - Double edge clocking sends two bits per cycle per pin
  - Different types distinguished by size of small prefetch buffer:
    - DDR (2 bits), DDR2 (4 bits), DDR4 (8 bits)
  - By 2010, standard for most server and desktop systems
  - Intel Core i7 supports only DDR3 SDRAM
Nonvolatile Memories

- DRAM and SRAM are volatile memories
  - Lose information if powered off.

- Nonvolatile memories retain value even if powered off
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Erasable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically erasable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs with partial (sector) erase capability
    - Wears out after about 100,000 erasing.

- Uses for Nonvolatile Memories
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems, …)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops, …)
  - Disk caches
What’s Inside A Disk Drive?

- Spindle
- Arm
- Actuator
- Platters
- Electronics (including a processor and memory!)
- SCSI connector

Image courtesy of Seagate Technology
Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.
Disk Capacity

- **Capacity**: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB), where $1 \text{ GB} = 10^9 \text{ Bytes}$

- Capacity is determined by these technology factors:
  - **Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - **Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - **Areal density** (bits/in²): product of recording and track density.

- Modern disks partition tracks into disjoint subsets called **recording zones**
  - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  - Each zone has a different number of sectors/track
Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x
(# tracks/surface) x (# surfaces/platter) x
(# platters/disk)

Example:
- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

Capacity = 512 x 300 x 20000 x 2 x 5
= 30,720,000,000
= 30.72 GB
Disk Access Time

- Average time to access some target sector approximated by:
  - $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$

- **Seek time** ($T_{\text{avg seek}}$)
  - Time to position heads over cylinder containing target sector.
  - Typical $T_{\text{avg seek}}$ is 3—9 ms

- **Rotational latency** ($T_{\text{avg rotation}}$)
  - Time waiting for first bit of target sector to pass under r/w head.
  - $T_{\text{avg rotation}} = \frac{1}{2} \times \frac{1}{\text{RPMs}} \times 60 \text{ sec/1 min}$
  - Typical $T_{\text{avg rotation}} = 7200$ RPMs

- **Transfer time** ($T_{\text{avg transfer}}$)
  - Time to read the bits in the target sector.
  - $T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{\text{avg # sectors/track}} \times 60 \text{ secs/1 min.}$
Disk Access Time Example

- **Given:**
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.

- **Derived:**
  - \[ T_{avg\ rotation} = \frac{1}{2} \times (60 \text{ secs/7200 RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms.} \]
  - \[ T_{avg\ transfer} = \frac{60}{7200} \text{ RPM} \times \frac{1}{400} \text{ secs/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms} \]
  - \[ T_{access} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms} \]

- **Important points:**
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower than DRAM.
Logical Disk Blocks

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)

- Mapping between logical blocks and actual (physical) sectors
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface, track, sector) triples.

- Allows controller to set aside spare cylinders for each zone.
  - Accounts for the difference in “formatted capacity” and “maximum capacity”.
Solid State Disks (SSDs)

- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after 100,000 repeated writes.
SSD Tradeoffs vs Rotating Disks

- **Advantages**
  - No moving parts \(\rightarrow\) faster, less power, more rugged

- **Disadvantages**
  - Have the potential to wear out
    - Mitigated by “wear leveling logic” in flash translation layer
    - E.g. Intel X25 guarantees 1 petabyte \((10^{15}\) bytes) of random writes before they wear out
  - In 2010, about 100 times more expensive per byte

- **Applications**
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers
## Storage Trends

### SRAM

<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>$/MB</td>
<td>19,200</td>
<td>2,900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>75</td>
<td>60</td>
<td>320</td>
</tr>
<tr>
<td>access (ns)</td>
<td>300</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>200</td>
</tr>
</tbody>
</table>

### DRAM

<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>$/MB</td>
<td>8,000</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>0.06</td>
<td>130,000</td>
</tr>
<tr>
<td>access (ns)</td>
<td>375</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>0.064</td>
<td>0.256</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>2,000</td>
<td>8,000</td>
<td>125,000</td>
</tr>
</tbody>
</table>

### Disk

<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>$/MB</td>
<td>500</td>
<td>100</td>
<td>8</td>
<td>0.30</td>
<td>0.01</td>
<td>0.005</td>
<td>0.0003</td>
<td>1,600,000</td>
</tr>
<tr>
<td>access (ms)</td>
<td>87</td>
<td>75</td>
<td>28</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>1</td>
<td>10</td>
<td>160</td>
<td>1,000</td>
<td>20,000</td>
<td>160,000</td>
<td>1,500,000</td>
<td>1,500,000</td>
</tr>
</tbody>
</table>

CEN591 Fall 2011
The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.

The graph shows the gap between DRAM, disk, and CPU speeds from 1980 to 2010. The y-axis represents time in nanoseconds (ns), and the x-axis represents years from 1980 to 2010.

- **Disk seek time** is represented by a filled diamond symbol.
- **Flash SSD access time** is represented by a filled triangle symbol.
- **DRAM access time** is represented by a filled square symbol.
- **SRAM access time** is represented by a filled circle symbol.
- **CPU cycle time** is represented by a filled diamond symbol with a square outline.
- **Effective CPU cycle time** is represented by an open circle symbol.

The gap between these components has widened over time, indicating a need for improved memory and data transfer speeds to match CPU performance.
Locality to the Rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality.
Locality

- **Principle of Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality**: Recently referenced items are likely to be referenced again in the near future.

- **Spatial locality**: Items with nearby addresses tend to be referenced close together in time.
Locality Example

Data references
- Reference array elements in succession (stride-1 reference pattern).
- Reference variable \( \text{sum} \) each iteration.

Instruction references
- Reference instructions in sequence.
- Cycle through loop repeatedly.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```
Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Question:** Does this function have good locality with respect to array `a`?

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Remember "Row-Major" ordering of array elements.
Locality Example

Question: Does this function have good locality with respect to array a?

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Remember row major recording
Locality Example

Question: Can you permute the loops so that the function scans the 3-d array $a$ with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[i][j][k];
    return sum;
}
```
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
An Example Memory Hierarchy

- CPU registers hold words retrieved from L1 cache
- L1 cache holds cache lines retrieved from L2 cache
- L2 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers
- Remote secondary storage (tapes, distributed file systems, Web servers)

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte
Caches

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- **Fundamental idea of a memory hierarchy**:
  - For each $k$, the faster, smaller device at level $k$ serves as a cache for the larger, slower device at level $k+1$.

- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level $k$ more often than they access the data at level $k+1$.
  - Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.

- **Big Idea**: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.
General Cache Concepts

Smaller, faster, more expensive memory caches a subset of the blocks.

Data is copied in block-sized transfer units.

Larger, slower, cheaper memory viewed as partitioned into “blocks”.

Cache

Memory
Data in block b is needed

Block b is in cache:
Hit!
Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)
Cache placement policy

- Not restricted placement: (any block at level k+1 stores at any block at level k)
  - High locating cost

- Restricted placement: A particular block of level K+1 can only reside at a small subset of the blocks at level k
  - Ex: a block i at level k+1 must be placed in block i mod 4 at level k
General Caching Concepts:
Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.

- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k (Restricted placement policy).
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time for previous example.

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
## Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-8 bytes words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>On/Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
Summary of caching

- The speed gap between CPU, memory and mass storage continues to widen.

- Well-written programs exhibit a property called locality.

- Memory hierarchies based on caching close the gap by exploiting locality.
Cache memory organization and operation
**System Cache Memories**

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in caches** (e.g., L1, L2, and L3), then in main memory.
- **Typical system structure:**

![Diagram of system structure with labeled components: CPU chip, Cache memories, Register file, ALU, Bus interface, I/O bridge, Memory bus, Main memory.]}
General Cache Organization (S, E, B)

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets
- \( B = 2^b \) bytes per cache block (the data)

Cache size:
\[ C = S \times E \times B \text{ data bytes} \]
Cache Read

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets

Address of word:
- \( t \) bits
- \( s \) bits
- \( b \) bits

- Tag
- Set index
- Block offset

Data begins at this offset

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

\( B = 2^b \) bytes per cache block (the data)

\( v \) valid bit
Quiz 6 review
Questions 1 and 2

Why are stacks required for procedure calls? What is the difference between the stacks in IA-32 and x86-64?

- To store the procedure caller state including return address, registers, and arguments
- X86-64 vs IA 32: less stack overhead
  - No stack frame pointer,
  - More registers to pass the arguments (up to 6 arguments can be passed by registers)
  - Different register convention

Which of the following statements are correct? (red colors are false)

- RISC reduces number of instructions for a program compared to CISC
- Harvard architecture provides safer protection of the codes compared to Von Neumann Architecture
- After 10 sequential call of 10 different procedures the stack space is increased by 10 times.
- The processor state (flags) only can be explicitly set or reset by test and comp instructions.
Question 3

In the following code, A and B are constants defined with \#define. What are the values of A and B?

typedef struct{
  char array[B];
  int t;
  short s[B];
  int u;
} str2;

typedef struct {
  short x[A][B];
  int y;
} str1;

void setVal(str1* p, str2 *q)
int v1=q->t;
int v2=q->u;
p->y=v1+v2;

1. movl 12(%ebp), %eax
2. movl 40(%eax), %edx
3. addl 12(%eax), %edx
4. movl 8(%ebp), %eax
5. movl %edx, 96(%eax)

array t s u

%eax=q q+B q+B+4 q+3B+4

3B+4=40, B=12
2AB=96, A=4

AB x y

%eax=P P+2AB

CEN591 Fall 2011
Question 4

- For the following variable (myrec) how many bytes you can save by changing the order of variable definition in the struct? (Alignment of the length 4 for int and 4 for the struct is required).

```c
typedef struct{
    char c;
    int d;
    char a[2];
} myrec[5];
```

Number of byte saving: 4*5 = 20
Question 5

For the following code enter an input string for the \texttt{read} procedure so that the program prints the following string: "you entered the number 5". (the address for the test procedure is 0x123)

```c
void main(){
    read();
    printf("you did not enter the correct string\n");
test(10);
}

void read(){
    char buf[2];
    gets(buf)
}

void test(int x){
    printf("you entered the number %d\n",x);
    exit(0);
}
```

Assume GCC generates the address 0x123 for the test procedure and generates the following code for the read procedure. (you can show the read procedure stack before and after calling gets)

<table>
<thead>
<tr>
<th>read:</th>
<th>Stack before calling gets</th>
<th>Stack after calling gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. pushl %ebp</td>
<td>Return address</td>
<td>00 00 00 05</td>
</tr>
<tr>
<td>2. movl %esp, %ebp</td>
<td>Old ebp</td>
<td>00 00 01 23</td>
</tr>
<tr>
<td>3. pushl %ebx</td>
<td>Old ebx</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>4. subl $8, %esp</td>
<td>ebp</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>5. leal -8(%ebp), %ebx</td>
<td></td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>6. movl %ebx, (%esp)</td>
<td></td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>7. call gets</td>
<td></td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>8. popl %ebx</td>
<td></td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>9. popl %ebp</td>
<td></td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>10. ret</td>
<td></td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

buf