Lecture 13: The Memory Hierarchy

- Memory Technologies
- The Memory Hierarchy
- The Locality Principle
- Basic Caching

Notes:
- Lab 5 due Thursday

http://xkcd.com/908/
Our “Computing Machine”

We need to fetch one instruction each cycle

Ultimately data is loaded from and results stored to memory
Memory Technologies

Technologies have vastly different tradeoffs between capacity, access latency, bandwidth, energy, and cost – ... and logically, different applications

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>1000s of bits</td>
<td>20 ps</td>
<td>$$$$$</td>
</tr>
<tr>
<td>SRAM</td>
<td>~10 KB-10 MB</td>
<td>1-10 ns</td>
<td>~$1000</td>
</tr>
<tr>
<td>DRAM</td>
<td>~10 GB</td>
<td>80 ns</td>
<td>~$10</td>
</tr>
<tr>
<td>Flash*</td>
<td>~100 GB</td>
<td>100 us</td>
<td>~$1</td>
</tr>
<tr>
<td>Hard disk*</td>
<td>~1 TB</td>
<td>10 ms</td>
<td>~$0.10</td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)
Static RAM (SRAM)

- **Address decoder**
- **Wordlines** (horizontal)
- **Bitlines** (vertical, two per cell)
- **SRAM cell**
- **Drivers**
- **8x6 SRAM array**
- **Sense amplifiers**
- **Data in**
- **Data out**

<table>
<thead>
<tr>
<th>Address</th>
<th>Data in</th>
<th>Data out</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
SRAM Cell

6-MOSFET (6T) cell:
- Two CMOS inverters (4 MOSFETs) forming a bistable element
- Two access transistors
SRAM Read

1. Drivers precharge all bitlines to Vdd (1), and leave them floating.
2. Address decoder activates one wordline.
3. Each cell in the activated word slowly pulls down one of the bitlines to GND (0).
4. Sense amplifiers sense change in bitline voltages, producing output data.
SRAM Write

1. Drivers set and hold bitlines to desired values (Vdd and GND for 1, GND and Vdd for 0)
2. Address decoder activates one wordline
3. Each cell in word is overpowered by the drivers, stores value

All transistors are carefully sized so that bitline GND overpowers cell Vdd, but bitline Vdd does not overpower cell GND (why?)
Multiported SRAMs

- SRAM so far can do either one read or one write/cycle
- We can do multiple reads and writes with multiple ports by adding one set of wordlines and bitlines per port
- Cost/bit? For N ports…
  - Wordlines: \( \frac{N}{2N} \)
  - Bitlines: \( \frac{2N}{2N} \)
  - Access FETs: \( \frac{2N}{2N} \)

- Wires often dominate area \( \rightarrow O(N^2) \) area!
Summary: SRAMs

- Array of $k \times b$ cells (k words, b cells per word)
- Cell is a bistable element + access transistors
  - Analog circuit with carefully sized transistors to allow reads and writes
- Read: Precharge bitlines, activate wordline, sense
- Write: Drive bitlines, activate wordline, overpower cells

- 6 MOSFETs/cell… can we do better?
  - What’s the minimum number of MOSFETs needed to store a single bit?
1T Dynamic RAM (DRAM) Cell

C in storage capacitor determined by:
- better dielectric
- more area
- thinner film

✓ ~20x smaller area than SRAM cell → Denser and cheaper!
✗ Problem: Capacitor leaks charge, must be refreshed periodically (~milliseconds)
DRAM Writes and Reads

- **Writes:** Drive bitline to Vdd or GND, activate wordline, charge or discharge capacitor

- **Reads:**
  1. Precharge bitline to Vdd/2
  2. Activate wordline
  3. Capacitor and bitline share charge
    - If capacitor was discharged, bitline voltage decreases slightly
    - If capacitor was charged, bitline voltage increases slightly
  4. Sense bitline to determine if 0 or 1
    - Issue: **Reads are destructive!** (charge is gone!)
    - So, data must be rewritten to cell at end of read
Summary: DRAM

• 1T DRAM cell: transistor + capacitor
• Smaller than SRAM cell, but destructive reads and capacitors leak charge
• DRAM arrays include circuitry to:
  – Write word again after every read (to avoid losing data)
  – Refresh (read+write) every word periodically

• DRAM vs SRAM:
  – ~20x denser than SRAM
  – ~2-10x slower than SRAM
Non-Volatile Storage: Flash

Flash Memory: Use “floating gate” transistors to store charge
- **Very dense**: Multiple bits/transistor, read and written in blocks
- **Slow** (especially on writes), 10-100 us
- **Limited number of writes**: charging/discharging the floating gate (writes) requires large voltages that damage transistor

Electrons here diminish strength of field from control gate ⇒ no inversion ⇒ NFET stays off even when word line is high.

Cyferz (CC BY 2.5)
Non-Volatile Storage: Hard Disk

Hard Disk: Rotating magnetic platters + read/write head
- Extremely slow (~10ms): Mechanically move head to position, wait for data to pass underneath head
- ~100MB/s for sequential read/writes
- ~100KB/s for random read/writes
- Cheap
Summary: Memory Technologies

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>1000s of bits</td>
<td>20 ps</td>
<td>$$$$$</td>
</tr>
<tr>
<td>SRAM</td>
<td>~10 KB-10 MB</td>
<td>1-10 ns</td>
<td>~$1000</td>
</tr>
<tr>
<td>DRAM</td>
<td>~10 GB</td>
<td>80 ns</td>
<td>~$10</td>
</tr>
<tr>
<td>Flash</td>
<td>~100 GB</td>
<td>100 us</td>
<td>~$1</td>
</tr>
<tr>
<td>Hard disk</td>
<td>~1 TB</td>
<td>10 ms</td>
<td>~$0.10</td>
</tr>
</tbody>
</table>

- Different technologies have vastly different tradeoffs
- Size is a **fundamental limit**, even setting cost aside:
  - Small + low latency, high bandwidth, low energy, **or**
  - Large + high-latency, low bandwidth, high energy
- Can we get the best of both worlds? (large, fast, cheap)
The Memory Hierarchy

Want large, fast, and cheap memory, but...
Large memories are slow (even if built with fast components)
Fast memories are expensive

Idea: Can we use a hierarchical system of memories with different tradeoffs to emulate a large, fast, cheap memory?

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>SRAM</th>
<th>DRAM</th>
<th>FLASH</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed:</td>
<td>Fastest</td>
<td>Slowest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td>Smallest</td>
<td>Largest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cost:</td>
<td>Highest</td>
<td>Lowest</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

≈

Fast
Large
Cheap
Memory Hierarchy Interface

Approach 1: Expose Hierarchy
- Registers, SRAM, DRAM, Flash, Hard Disk each available as storage alternatives
- Tell programmers: “Use them cleverly”

Approach 2: Hide Hierarchy
- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns
The Locality Principle

Keep the most often-used data in a small, fast SRAM (often local to CPU chip)

Refer to Main Memory only rarely, for remaining data.

The reason this strategy works: LOCALITY

Locality of Reference:
Access to address X at time t implies that access to address X+ΔX at time t+Δt becomes more probable as ΔX and Δt approach zero.
Memory Reference Patterns

S is the set of locations accessed during $\Delta t$.

Working set: a set $S$ which changes slowly wrt access time.

Working set size, $|S|$
Caches

Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations

- Very fast access if data is cached, otherwise accesses slower, larger cache or memory
- Exploits the locality principle

Computer systems often use multiple levels of caches

Caching widely applied beyond hardware (e.g., web caches)
A Typical Memory Hierarchy

- Everything is a cache for something else…

<table>
<thead>
<tr>
<th>Access time</th>
<th>Capacity</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 cycle</td>
<td>1 KB</td>
<td>Software/Compiler</td>
</tr>
<tr>
<td>2-4 cycles</td>
<td>32 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>10 cycles</td>
<td>256 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>40 cycles</td>
<td>10 MB</td>
<td>Hardware</td>
</tr>
<tr>
<td>200 cycles</td>
<td>10 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>10-100us</td>
<td>100 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>10ms</td>
<td>1 TB</td>
<td>Software/OS</td>
</tr>
</tbody>
</table>

On the datapath
- Registers
- Level 1 Cache
- Level 2 Cache
- Level 3 Cache

On chip
- Main Memory
- Flash Drive
- Hard Disk

Other chips
- Main Memory
- Flash Drive
- Hard Disk

Mechanical devices
- Main Memory
- Flash Drive
- Hard Disk

6.004 Computation Structures
A Typical Memory Hierarchy

- Everything is a cache for something else...

### A Typical Memory Hierarchy:

<table>
<thead>
<tr>
<th>Level</th>
<th>Access Time</th>
<th>Capacity</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1 cycle</td>
<td>1 KB</td>
<td>Software/Compiler</td>
</tr>
<tr>
<td>Level 1 Cache</td>
<td>2-4 cycles</td>
<td>32 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>Level 2 Cache</td>
<td>10 cycles</td>
<td>256 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>Level 3 Cache</td>
<td>40 cycles</td>
<td>10 MB</td>
<td>Hardware</td>
</tr>
<tr>
<td>Main Memory</td>
<td>200 cycles</td>
<td>10 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>Flash Drive</td>
<td>10-100us</td>
<td>100 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>10 ms</td>
<td>1 TB</td>
<td>Software/OS</td>
</tr>
</tbody>
</table>

**TODAY:** Hardware Caches

**LATER:** Software Caches (Virtual Memory)

**HW vs SW caches:**
- Same objective: fake large, fast, cheap mem
- Conceptually similar
- Different implementations (very different tradeoffs!)
Cache Access

• Processor sends address to cache
• Two options:
  – **Cache hit**: Data for this address in cache, returned quickly
  – **Cache miss**: Data not in cache
    • Fetch data from memory, send it back to processor
    • Retain this data in the cache (replacing some other data)
  – Processor must deal with variable memory access time

LD 0x6004
LD 0x6034
Cache Metrics

Hit Ratio: \[ HR = \frac{\text{hits}}{\text{hits} + \text{misses}} = 1 - MR \]

Miss Ratio: \[ MR = \frac{\text{misses}}{\text{hits} + \text{misses}} = 1 - HR \]

Average Memory Access Time (AMAT):

\[ AMAT = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty} \]

- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

\[ AMAT = \text{HitTime}_{L1} + \text{MissRatio}_{L1} \times AMAT_{L2} = \]
\[ AMAT = \text{HitTime}_{L1} + \text{MissRatio}_{L1} \times (\text{HitTime}_{L2} + \text{MissRatio}_{L2} \times AMAT_{L3}) = ... \]
Example: How High of a Hit Ratio?

What hit ratio do we need to break even? (Main memory only: AMAT = 100)

\[ 100 = 4 + (1 - HR) \times 100 \quad \Rightarrow \quad HR = 4\% \]

What hit ratio do we need to achieve AMAT = 5 cycles?

\[ 5 = 4 + (1 - HR) \times 100 \quad \Rightarrow \quad HR = 99\% \]
Basic Cache Algorithm

ON REFERENCE TO Mem[X]:
Look for X among cache tags...

HIT: $X = TAG(i)$, for some cache line $i$
- READ: return $DATA(i)$
- WRITE: change $DATA(i)$; Start Write to Mem(X)

MISS: $X$ not found in TAG of any cache line

- REPLACEMENT SELECTION:
  Select some line $k$ to hold Mem[X] (Allocation)

  - READ: Read Mem[X]
    Set TAG($k$)=$X$, DATA($k$)=Mem[X]

  - WRITE: Start Write to Mem(X)
    Set TAG($k$)=$X$, DATA($k$)= new Mem[X]

Q: How do we “search” the cache?