CPU clock rates ~0.2ns – 2ns (5GHz-500MHz)

<table>
<thead>
<tr>
<th>Technology</th>
<th>Capacity</th>
<th>$/GB</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tape</td>
<td>1 TB</td>
<td>$.17</td>
<td>100s of seconds</td>
</tr>
<tr>
<td>Disk</td>
<td>1 TB</td>
<td>$.08</td>
<td>Millions cycles (ms)</td>
</tr>
<tr>
<td>SSD (Flash)</td>
<td>128GB</td>
<td>$3</td>
<td>Thousands of cycles (us)</td>
</tr>
<tr>
<td>DRAM</td>
<td>4GB</td>
<td>$25</td>
<td>50-300 cycles (10s of ns)</td>
</tr>
<tr>
<td>SRAM off-chip</td>
<td>4MB</td>
<td>$4k</td>
<td>5-15 cycles (few ns)</td>
</tr>
<tr>
<td>SRAM on-chip</td>
<td>256 KB</td>
<td>???</td>
<td>1-3 cycles (ns)</td>
</tr>
</tbody>
</table>

Others: **eDRAM** aka **1-T SRAM**, FeRAM, CD, DVD, ...

Q: Can we create illusion of cheap + large + fast?
Memory Pyramid

- **Disk** (Many GB – few TB) - 1000000+ cycle access
- **Memory** (128MB – few GB) - 50-300 cycle access
- **L2 Cache** (½-32MB) - 5-15 cycle access
- **L1 Cache** (several KB) - 1-3 cycle access
- **RegFile** - 1 cycle access

L3 becoming more common (eDRAM ?)

These are rough numbers: mileage may vary for latest/greatest Caches usually made of SRAM (or eDRAM)
Memory closer to processor

• small & fast
• stores active data

Memory farther from processor

• big & slow
• stores inactive data
Assumption: Most data is not active.
Q: How to decide what is active?
A: Some committee decides
A: Programmer decides
A: Compiler decides
A: OS decides at run-time
A: Hardware decides at run-time
Q: What is “active” data?
A: Data that will be used soon.

If Mem[x] is was accessed recently...

... then Mem[x] is likely to be accessed soon

• Caches exploit temporal locality by putting recently accessed Mem[x] higher in the pyramid

... then Mem[x ± ε] is likely to be accessed soon

• Caches exploit spatial locality by putting an entire block containing Mem[x] higher in the pyramid
int n = 4;
int k[] = { 3, 14, 0, 10 };

int fib(int i) {
    if (i <= 2) return i;
    else return fib(i-1)+fib(i-2);
}

int main(int ac, char **av) {
    for (int i = 0; i < n; i++) {
        printf(fib(k[i]));
        printf("\n");
    }
}
Memory closer to processor is fast and small

• usually stores **subset** of memory farther from processor
  – “strictly inclusive”

• alternatives:
  – strictly exclusive
  – mostly inclusive

• Transfer whole **blocks**
  **cache lines**, e.g:
  4kb: disk ↔ ram
  256b: ram ↔ L2
  64b: L2 ↔ L1
Processor tries to access Mem[x]
Check: is block containing x in the cache?

• Yes: cache hit
  – return requested data from cache line

• No: cache miss
  – read block from memory (or lower level cache)
  – (evict an existing cache line to make room)
  – place new block in cache
  – return requested data

→ and stall the pipeline while all of this happens
Cache has to be **fast and dense**

- Gain speed by performing lookups in parallel
  - but requires die real estate for lookup logic
- Reduce lookup logic by limiting where in the cache a block might be placed
  - but might reduce cache effectiveness
A given data block can be placed...

- ... in any cache line ➔ Fully Associative
- ... in exactly one cache line ➔ Direct Mapped
- ... in a small set of cache lines ➔ Set Associative
Direct Mapped Cache

- Each block number mapped to a single cache line index
- Simplest hardware

<table>
<thead>
<tr>
<th>line 0</th>
<th>line 1</th>
<th>line 2</th>
<th>line 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
0x000000 0x000004 0x000008 0x00000c 0x000010 0x000014 0x000018 0x00001c 0x000020 0x000024 0x000028 0x00002c 0x000030 0x000034 0x000038 0x00003c 0x000040 0x000044 0x000048 0x00004c
```
Assume sixteen 64-byte cache lines
0x7FFF3D4D
  = 0111 1111 1111 1111 0011 1101 0100 1101

Need meta-data for each cache line:
  • valid bit: is the cache line non-empty?
  • tag: which block is stored in this line (if valid)

Q: how to check if X is in the cache?
Q: how to clear a cache line?
$n$ bit index, $m$ bit offset

Q: How big is cache (data only)?

Q: How much SRAM needed (data + overhead)?
Using **byte addresses** in this example! Addr Bus = 5 bits

### Processor
- `lb $1 ← M[ 1 ]`
- `lb $2 ← M[ 13 ]`
- `lb $3 ← M[ 0 ]`
- `lb $3 ← M[ 6 ]`
- `lb $2 ← M[ 5 ]`
- `lb $2 ← M[ 6 ]`
- `lb $2 ← M[ 10 ]`
- `lb $2 ← M[ 12 ]`

### Memory

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| A | 101| 103| 107| 109| 113| 127| 131| 137| 139| 149| 151| 157| 163| 167| 173| 179| 181|

### Direct Mapped Cache

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hits:**

**Misses:**
Using **byte addresses** in this example! Addr Bus = 5 bits

### Processor

<table>
<thead>
<tr>
<th>$lb$</th>
<th>(M[1])</th>
<th>(M[13])</th>
<th>(M[0])</th>
<th>(M[6])</th>
<th>(M[5])</th>
<th>(M[6])</th>
<th>(M[10])</th>
<th>(M[12])</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>103</td>
</tr>
<tr>
<td>2</td>
<td>107</td>
</tr>
<tr>
<td>3</td>
<td>109</td>
</tr>
<tr>
<td>4</td>
<td>113</td>
</tr>
<tr>
<td>5</td>
<td>127</td>
</tr>
<tr>
<td>6</td>
<td>131</td>
</tr>
<tr>
<td>7</td>
<td>137</td>
</tr>
<tr>
<td>8</td>
<td>139</td>
</tr>
<tr>
<td>9</td>
<td>149</td>
</tr>
<tr>
<td>10</td>
<td>151</td>
</tr>
<tr>
<td>11</td>
<td>157</td>
</tr>
<tr>
<td>12</td>
<td>163</td>
</tr>
<tr>
<td>13</td>
<td>167</td>
</tr>
<tr>
<td>14</td>
<td>173</td>
</tr>
<tr>
<td>15</td>
<td>179</td>
</tr>
<tr>
<td>16</td>
<td>181</td>
</tr>
</tbody>
</table>

### Direct Mapped Cache

\[ A = \]

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hits:** 0  
**Misses:** 0
Direct Mapped Cache (Reading)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Tag</td>
<td>Block</td>
</tr>
</tbody>
</table>

Diagram: Table with columns Tag, Index, Offset, and rows V, Tag, Block.
Direct Mapped Cache Size

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

$n$ bit index, $m$ bit offset

Q: How big is cache (data only)?
Q: How much SRAM needed (data + overhead)?
Cache Performance (very simplified):

**L1 (SRAM):** 512 x 64 byte cache lines, direct mapped
- Data cost: 3 cycle per word access
- Lookup cost: 2 cycle

**Mem (DRAM):** 4GB
- Data cost: 50 cycle per word, plus 3 cycle per consecutive word

**Performance depends on:**
- Access time for hit, miss penalty, hit rate
Cache misses: classification

The line is being referenced for the first time

- **Cold (aka Compulsory) Miss**

The line was in the cache, but has been evicted
Q: How to avoid...

**Cold Misses**

- Unavoidable? The data was never in the cache...
- Prefetching!

**Other Misses**

- Buy more SRAM
- Use a more flexible cache design
Bigger cache doesn’t always help...
Memcpy access trace: 0, 16, 1, 17, 2, 18, 3, 19, 4, 20, ...
Hit rate with four direct-mapped 2-byte cache lines?

With eight 2-byte cache lines?

With four 4-byte cache lines?
Cache misses: classification

The line is being referenced for the first time

• Cold (aka Compulsory) Miss

The line was in the cache, but has been evicted...

... because some other access with the same index

• Conflict Miss

... because the cache is too small

• i.e. the \textit{working set} of program is larger than the cache

• Capacity Miss
Q: How to avoid...

Cold Misses

• Unavoidable? The data was never in the cache...
• Prefetching!

Capacity Misses

• Buy more SRAM

Conflict Misses

• Use a more flexible cache design
A given data block can be placed...

- ... in any cache line ➔ **Fully Associative**
- ... in exactly one cache line ➔ **Direct Mapped**
- ... in a small set of cache lines ➔ **Set Associative**
Using **byte addresses** in this example! Addr Bus = 5 bits

### Processor
- `lb $1 \leftarrow M[1]`
- `lb $2 \leftarrow M[13]`
- `lb $3 \leftarrow M[0]`
- `lb $3 \leftarrow M[6]`
- `lb $2 \leftarrow M[5]`
- `lb $2 \leftarrow M[6]`
- `lb $2 \leftarrow M[10]`
- `lb $2 \leftarrow M[12]`
- $1$
- $2$
- $3$
- $4$

### Memory
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>103</td>
<td>107</td>
<td>109</td>
<td>113</td>
<td>127</td>
<td>131</td>
<td>137</td>
<td>139</td>
<td>149</td>
<td>151</td>
<td>157</td>
<td>163</td>
<td>167</td>
<td>173</td>
<td>179</td>
<td>181</td>
</tr>
</tbody>
</table>

### Fully Associative Cache

- **V**
- **tag**
- **data**

<table>
<thead>
<tr>
<th>Hits:</th>
<th>Misses:</th>
<th>A =</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A Simple Fully Associative Cache
Fully Associative Cache (Reading)

- **Tag**
- **Offset**

- **V Tag**
- **Block**

- **hit?**

- **line select**
  - **64bytes**

- **word select**
  - **32bits**

- **data**
### Fully Associative Cache Size

<table>
<thead>
<tr>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
</table>

$m$ bit offset, $2^n$ cache lines

Q: How big is cache (data only)?

Q: How much SRAM needed (data + overhead)?
Fully-associative reduces conflict misses...
... assuming good eviction strategy
Memcpy access trace: 0, 16, 1, 17, 2, 18, 3, 19, 4, 20, ...
Hit rate with four fully-associative 2-byte cache lines?
... but large block size can still reduce hit rate
vector add access trace: 0, 100, 200, 1, 101, 201, 2, 202, ...
Hit rate with four fully-associative 2-byte cache lines?

With two 4-byte cache lines?
Cache misses: classification

Cold (aka Compulsory)

• The line is being referenced for the first time

Capacity

• The line was evicted because the cache was too small
• i.e. the *working set* of program is larger than the cache

Conflict

• The line was evicted because of another access whose index conflicted
Caching assumptions

• small working set: 90/10 rule
• can predict future: spatial & temporal locality

Benefits

• big & fast memory built from (big & slow) + (small & fast)

Tradeoffs:

associativity, line size, hit cost, miss penalty, hit rate

• Fully Associative $\rightarrow$ higher hit cost, higher hit rate
• Larger block size $\rightarrow$ lower hit cost, higher miss penalty

Next up: other designs; writing to caches