The Memory Hierarchy

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Memory Technologies

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>100s 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>~$0.1</td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)
Memory Technologies

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost
  - ... and thus 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>100s 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>~$0.1</td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)
Memory Technologies

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost
  - ... and thus 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>100s 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>~$0.1</td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)
Memory Technologies

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost.
  - ... and thus different applications

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost/GB</th>
<th>Processor</th>
<th>Datapath</th>
<th>Memory Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>100s of bits</td>
<td>20 ps</td>
<td>$$$$$</td>
<td>Processor</td>
<td>Datapath</td>
<td>Memory Hierarchy</td>
</tr>
<tr>
<td>SRAM</td>
<td>~10 KB-10 MB</td>
<td>1-10 ns</td>
<td>~$1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>~10 GB</td>
<td>80 ns</td>
<td>~$10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash*</td>
<td>~100 GB</td>
<td>100 us</td>
<td>~$1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hard disk*</td>
<td>~1 TB</td>
<td>10 ms</td>
<td>~$0.1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)
Memory Technologies

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost
  - ... and thus 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>100s 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>~$0.1</td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)
Memory Technologies: SRAM, DRAM, Flash

NOTE: Demystification, will not be on the quiz
Static RAM (SRAM)

Drivers

Address decoder

8x6 SRAM array

Sense amplifiers

April 3, 2018
Static RAM (SRAM)
Static RAM (SRAM)

Drivers

Wordlines (horizontal)

Sense amplifiers

Address decoder

8x6 SRAM array

Address 3
Static RAM (SRAM)

Drivers

SRAM cell

Wordlines (horizontal)

Address decoder

Address

3

8x6 SRAM array

Sense amplifiers

April 3, 2018
Static RAM (SRAM)

Drivers

SRAM cell

Wordlines (horizontal)

Bitlines (vertical)

Address decoder

8x6 SRAM array

Sense amplifiers
Static RAM (SRAM)

- Drivers
- Sense amplifiers
- Address decoder
- SRAM cell
- Wordlines (horizontal)
- Bitlines (vertical)
- 8x6 SRAM array
- Data out

8x6 SRAM array

April 3, 2018
Static RAM (SRAM)

Drivers

SRAM cell

Wordlines (horizontal)

Bitlines (vertical)

Address decoder

Address

8x6 SRAM array

Data in

6

Data in

3

Data out

6
SRAM Cell

6-transistor (6T) cell:
SRAM Cell

6-transistor (6T) cell:
- Two CMOS inverters (4 FETs) forming a bistable element

Bistable element (two stable states) stores a single bit

Vdd \[\rightarrow\] GND “1”

GND \[\rightarrow\] Vdd “0”
SRAM Cell

6-transistor (6T) cell:
- Two CMOS inverters (4 FETs) forming a bistable element
- Two access transistors

Bistable element (two stable states) stores a single bit

Vdd GND “1”
GND Vdd “0”
SRAM Read

6T SRAM Cell

access FETs

bitline

wordline

April 3, 2018
1. Drivers precharge all bitlines to Vdd (1), and leave them floating.
1. Drivers precharge all bitlines to Vdd (1), and leave them floating.

2. Address decoder activates one wordline.
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).
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, produce output data
SRAM Write

bitline

Vdd

GND

wordline

access FETs
SRAM Write

1. Drivers set and hold bitlines to desired values (Vdd and GND for 1, GND and Vdd for 0)
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
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
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

Cell transistors are carefully sized so that bitline GND overpowers cell Vdd, but bitline Vdd does not overpower cell GND
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
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
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
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
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
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?
  - Bitlines?
  - Access FETs?
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? \( N \)
  - Bitlines?
  - Access FETs?
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? $N$
  - Bitlines? $2N$
  - Access FETs?
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? $N$
  - Bitlines? $2*N$
  - Access FETs? $2*N$
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? \( N \)
  - Bitlines? \( 2N \)
  - Access FETs? \( 2N \)

- Wires 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
- Read: Precharge bitlines, activate wordline, sense
- Write: Drive bitlines, activate wordline, overpower cells
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
- Read: Precharge bitlines, activate wordline, sense
- Write: Drive bitlines, activate wordline, overpower cells

- 6 FETs/cell... can we do better?
1T Dynamic RAM (DRAM) Cell

- **wordline**
- **access FET**
- **bitline**
1T Dynamic RAM (DRAM) Cell

Storage capacitor

1T DRAM Cell

\( V_{\text{REF}} \)

wordline

access FET

bitline
1T Dynamic RAM (DRAM) Cell

Storage capacitor

1T DRAM Cell

wordline

access FET

bitline

Trench capacitors take little area

Cyferz (CC BY 2.5)
1T Dynamic RAM (DRAM) Cell

1T DRAM Cell

- Storage capacitor
- Wordline
- Access FET
- Bitline

Trench capacitors take little area

Cyferz (CC BY 2.5)
1T Dynamic RAM (DRAM) Cell

- **Storage capacitor**
- **1T DRAM Cell**
- **V_{REF}**
- **wordline**
- **access FET**
- **bitline**

- Trench capacitors take little area

✓~20x smaller area than SRAM cell → Denser and cheaper!
1T Dynamic RAM (DRAM) Cell

Storage capacitor

1T DRAM Cell

V_{REF}

wordline

access FET

bitline

Trench capacitors take little area

✓ ~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

![Diagram of 1T DRAM Cell]

- bitline
- word line
- Storage capacitor
- \( V_{\text{REF}} \)
- access FET

April 3, 2018
DRAM Writes and Reads

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

• Reads:
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
**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
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!)
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!)  
  - Data must be rewritten to cells at end of read
Non-Volatile Storage: Flash

Flash Memory: Use “floating gate” transistors to store charge

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: Flash

Flash Memory: Use “floating gate” transistors to store charge

- Very dense: Multiple bits/transistor, read and written in blocks

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: 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

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: 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)
The Memory Hierarchy
## 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>100s 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>~$0.1</td>
</tr>
</tbody>
</table>

- Different technologies have vastly different tradeoffs
Summary: Memory Technologies

- 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

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

- 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 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

Solution: Use a hierarchy of memories with different tradeoffs to fake a large, fast, cheap memory
Memory Hierarchy Interface

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

\[ \text{address} \]

\[ \text{time} \]
Typical Memory Access Patterns

address

code

time
Typical Memory Access Patterns

![Diagram showing memory access patterns over time and address with loops highlighted.]

- address
- time
- code
- loop

April 3, 2018 L13-17
Typical Memory Access Patterns
Typical Memory Access Patterns

- **address**
  - stack accesses
  - local variable accesses

- **code**
  - loop

- **time**
Typical Memory Access Patterns

- **stack**
- **code**
- **address**
- **time**

- Local variable accesses
- Procedure calls
- Loop
Typical Memory Access Patterns

- **address**
  - data
  - stack
  - code

- **time**
  - loop
  - procedure calls
  - local variable accesses
Typical Memory Access Patterns

- **address**
- **data**
- **stack**
- **code**

- **array accesses**
- **local variable accesses**
- **procedure calls**
- **loop**

April 3, 2018
Common Predictable Patterns

- Two predictable properties of memory accesses:
  - **Temporal locality**: If a location has been accessed recently, it is likely to be accessed (reused) soon
  - **Spatial locality**: If a location has been accessed recently, it is likely that nearby locations will be accessed soon
Caches

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

```
CPU          Cache          Main Memory
 Address     | Address     Data ─── Data
 │            │            ┌─────┐
 └───────────┘          │          └─────┘
```

April 3, 2018
Caches

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

![Diagram showing interaction between CPU, Cache, and Main Memory]

- Processor sends accesses to cache. Two options:
Caches

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

• Processor sends accesses to cache. Two options:
  – Cache hit: Data for this address in cache, returned quickly
Caches

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

- Processor sends accesses 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)
Caches

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

- **Processor** sends accesses 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.
A Typical Memory Hierarchy

Computers use many levels of caches:

<table>
<thead>
<tr>
<th>On the CPU</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>10ms</td>
<td>1 TB</td>
<td>Software/OS</td>
</tr>
</tbody>
</table>

On chip

Other chips and devices

April 3, 2018
Cache Metrics

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

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

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

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

- **Average Memory Access Time (AMAT):**

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

  - Goal of caching is to improve AMAT
Cache Metrics

- **Hit Ratio**: \( HR = \frac{\text{hits}}{\text{hits} + \text{misses}} = 1 \quad MR \)

- **Miss Ratio**: \( MR = \frac{\text{misses}}{\text{hits} + \text{misses}} = 1 \quad HR \)

- **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)
Example: How High of a Hit Ratio?

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

\[ 100 = 4 + (1 - HR) \times 100 \Rightarrow HR = 4\% \]
Example: How High of a Hit Ratio?

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

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

What hit ratio do we need to achieve AMAT = 5 cycles?
Example: How High of a Hit Ratio?

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

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

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

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

On reference to Mem[X], look for X among cache tags

April 3, 2018
Basic Cache Algorithm (Reads)

On reference to Mem[X], look for X among cache tags.

HIT: X = Tag(i) for some cache line i
Basic Cache Algorithm (Reads)

On reference to Mem[X],
look for X among cache tags

HIT: X = Tag(i)
for some cache line i

Return Data(i)
Basic Cache Algorithm (Reads)

On reference to Mem[X], look for X among cache tags
- **HIT:** X = Tag(i) for some cache line i
- **MISS:** X not found in Tag of any cache line

Return Data(i)
Basic Cache Algorithm (Reads)

On reference to Mem[X], look for X among cache tags

HIT: X = Tag(i) for some cache line i

MISS: X not found in Tag of any cache line

Return Data(i)

Read Mem[X]
Return Mem[X]
Select a line k to hold Mem[X]
Write Tag(k)=X, Data(k) = Mem[X]
**Basic Cache Algorithm (Reads)**

On reference to Mem[X], look for X among cache tags

- **HIT**: \( X = \text{Tag}(i) \) for some cache line \( i \)
  - Return Data(i)

- **MISS**: X not found in Tag of any cache line
  - Read Mem[X]
  - Return Mem[X]
  - Select a line \( k \) to hold Mem[X]
  - Write Tag(k) = X, Data(k) = Mem[X]

**Q: How do we “search” the cache?**
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT

- Example 8-line direct-mapped cache:

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (27 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit BYTE address

```
0000000000000000000000000000000011101000
```
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the **index bits**)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT

**Example 8-line direct-mapped cache:**

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (27 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit BYTE address

0000000000000000000000000000000011101000

**Offset bits**
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT

- Example 8-line direct-mapped cache:

```
32-bit BYTE address
00000000000000000000000001101000
```

```
Valid bit | Tag (27 bits) | Data (32 bits)
---|---|---
0 | |||
1 | |||
3 | |||
4 | |||
5 | |||
6 | |||
7 | |||
```

Index bits: 0-2
Offset bits: 3-6

Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT

- Example 8-line direct-mapped cache:

```
32-bit BYTE address
00000000000000000000000000000000111101000
```

```
Valid bit     Tag (27 bits)     Data (32 bits)
0             00000000000000000000000000000000
1             00000000000000000000000000000000
3             00000000000000000000000000000000
4             00000000000000000000000000000000
5             00000000000000000000000000000000
6             00000000000000000000000000000000
7             00000000000000000000000000000000
```

Index: 011101000
Offset: 00000000000000000000000000000000

April 3, 2018 L13-24
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit $== 1$ and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:

```
32-bit BYTE address
00000000000000000000000000011101000

Index bits
Tag bits
Offset bits
=?
```

```
Valid bit
Tag (27 bits)
Data (32 bits)
```

Index into cache with 3 bits, read valid bit, tag, and data.
If valid bit $== 1$ and tag matches, HIT.

Diagram showing how address is divided into index, tag, and offset bits.
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT

Example 8-line direct-mapped cache:
Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

**Read Mem[0x400C]**

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x0000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x0000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>
# Example: Direct-Mapped Caches

A 64-line direct-mapped cache has 64 indexes, which require 6 index bits.

**Read Mem[0x400C]**

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>
Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100
TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>1 0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>1 0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>1 0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0 0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>1 0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>
# Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

**Read Mem[0x400C]**

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x0000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x0000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>

TAG: 0x40  
INDEX: 0x3  
OFFSET: 0x0
Example: Direct-Mapped Caches

64-line direct-mapped cache $\rightarrow$ 64 indexes $\rightarrow$ 6 index bits

**Read Mem[0x400C]**

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

HIT, DATA 0x42424242

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1 1</td>
<td>0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2 1</td>
<td>0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3 1</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4 0</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>63 1</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>
Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

**Read Mem[0x400C]**

- **TAG:** 0x40
- **INDEX:** 0x3
- **OFFSET:** 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>
Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

Read Mem[0x400C]

```
0100 0000 0000 1100
```

TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

INDEX: 0x2 → tag mismatch → MISS

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>

April 3, 2018 L13-25
Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

INDEX: 0x2 → tag mismatch
→ MISS

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>

Part of the address (index bits) is encoded in the location Tag + Index bits unambiguously identify the data’s address
### Block Size

- **Take advantage of spatial locality:** Store multiple words per data block
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer blocks in the cache

- **Example:** 4-block, 16-word direct-mapped cache

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (26 bits)</th>
<th>Data (4 words, 16 bytes)</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>
</tbody>
</table>

32-bit BYTE address
Block Size

- Take advantage of spatial locality: Store multiple words per data block
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer blocks in the cache

- Example: 4-block, 16-word direct-mapped cache

![Diagram showing a 32-bit BYTE address and a 4-block, 16-word direct-mapped cache with block offset bits: 4 (16 bytes/block).]
Block Size

- Take advantage of spatial locality: Store multiple words per data block
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer blocks in the cache

- Example: 4-block, 16-word direct-mapped cache

```
32-bit BYTE address
```

```
<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (26 bits)</th>
<th>Data (4 words, 16 bytes)</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>
</tbody>
</table>
```

Index bits: 2 (4 indexes)

Block offset bits: 4 (16 bytes/block)
Block Size

- Take advantage of spatial locality: Store multiple words per data block
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer blocks in the cache

- Example: 4-block, 16-word direct-mapped cache

```
| Valid bit | Tag (26 bits) | Data (4 words, 16 bytes) |
```

- Tag bits: 26 (=32-4-2)
- Index bits: 2 (4 indexes)
- Block offset bits: 4 (16 bytes/block)
Block Size Tradeoffs

• Larger block sizes...
  – Take advantage of spatial locality
  – Incur larger miss penalty since it takes longer to transfer the block from memory
  – Can increase the average hit time and miss ratio
Block Size Tradeoffs

- Larger block sizes...
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio

![Miss Penalty vs Block Size Graph]
Block Size Tradeoffs

• Larger block sizes...
  – Take advantage of spatial locality
  – Incur larger miss penalty since it takes longer to transfer the block from memory
  – Can increase the average hit time and miss ratio

![Graph showing tradeoffs between block size and miss penalty/miss ratio](image-url)
Block Size Tradeoffs

- Larger block sizes...
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio

- AMAT = HitTime + MissPenalty*MissRatio

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Miss Penalty</th>
<th>Miss Ratio</th>
<th>AMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>~64 bytes</td>
<td>Exploits spatial locality</td>
<td>Few blocks, compromises temporal locality</td>
<td>Increased miss penalty and miss rate</td>
</tr>
</tbody>
</table>

April 3, 2018
**Direct-Mapped Cache Problem: Conflict Misses**

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>HIT</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>HIT</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>HIT</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>HIT</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume:
- 1024-line DM cache
- Block size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing
Direct-Mapped Cache Problem: Conflict Misses

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>HIT</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>HIT</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>HIT</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>HIT</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
</tbody>
</table>

Assume:
- 1024-line DM cache
- Block size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>MISS</td>
</tr>
<tr>
<td>2049</td>
<td>1</td>
<td>MISS</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>MISS</td>
</tr>
<tr>
<td>2050</td>
<td>2</td>
<td>MISS</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
</tr>
</tbody>
</table>

Loop A: Code at 1024, data at 37

Loop B: Code at 1024, data at 2048
Direct-Mapped Cache Problem: Conflict Misses

**Loop A:**
- Code at 1024, data at 37

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/ Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>HIT</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>HIT</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>HIT</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>HIT</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inflexible mapping
(each address can only be in one cache location) \(\rightarrow\) Conflict misses!

**Loop B:**
- Code at 1024, data at 2048

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/ Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>MISS</td>
</tr>
<tr>
<td>2049</td>
<td>1</td>
<td>MISS</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>MISS</td>
</tr>
<tr>
<td>2050</td>
<td>2</td>
<td>MISS</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume:
- 1024-line DM cache
- Block size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = # ways
      = “set associativity” (e.g., 4-way → 4 lines/set)
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = # ways
      - = “set associativity”
        - (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = #ways = “set associativity” (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
  - Tags from all ways are checked in parallel
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = #ways
      = “set associativity”
      (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
  - Tags from all ways are checked in parallel

- Fully-associative cache: Extreme case with a single set and as many ways as lines
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = #ways = “set associativity” (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
  - Tags from all ways are checked in parallel

- Fully-associative cache: Extreme case with a single set and as many ways as lines
  - Any address can be in any line → No conflict misses, but expensive
N-way Set-Associative Cache

Example: 3-way 8-set cache

INCOMING ADDRESS

Tag Data

Tag Data

Tag Data

MEM DATA

DATA TO CPU

HIT
N-way Set-Associative Cache

Example: 3-way 8-set cache

INCOMING ADDRESS

Tag Data

Tag Data

Tag Data

SET

MEM DATA

DATA TO CPU

HIT
N-way Set-Associative Cache

Example: 3-way 8-set cache
Thank you!

Next lecture: Implementing Caches