CS 33

Memory Hierarchy I
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; 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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>Maybe</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>

- EDC = error detection and correction
  - to cope with noise, etc.
Conventional DRAM Organization

- **d x w** DRAM:
  - dw total bits organized as d **supercells** of size w bits
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)

Memory controller

64-bit doubleword at main memory address A

64-bit doubleword
Enhanced DRAMs

• Basic DRAM cell has not changed since its invention in 1966
  – commercialized by Intel in 1970

• DRAMs 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)
    » DDR1
      • twice as fast
    » DDR2
      • four times as fast
    » DDR3
      • eight times as fast
Enhanced DRAMs

SDR: n B/sec

DDR1: 2n B/sec

DDR2: 4n B/sec

DDR3: 8n B/sec
Quiz 1

A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR3 system $n$ times faster than on an SDR system, where $n$ is:

a) 1  
b) 2  
c) 4  
d) 8
Traditional Bus Structure Connecting CPU and Memory

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.
Memory Read Transaction (1)

- CPU places address A on the memory bus

Load operation: \texttt{movq A, %rax}

Diagram:
- Register file
- ALU
- Bus interface
- I/O bridge
- Main memory
- Address A
- Load operation: movq A, %rax
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus

Load operation: `movq A, %rax`
Memory Read Transaction (3)

- CPU reads word x from the bus and copies it into register %rax

Load operation: `movq A, %rax`
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Store operation: `movq %rax, A`
Memory Write Transaction (2)

- CPU places data word y on the bus

Store operation: `movq %rax, A`
Memory Write Transaction (3)

- Main memory reads data word \( y \) from the bus and stores it at address \( A \)

Store operation: \texttt{movq}\%rax, A
A Mismatch

• A processor clock cycle is ~0.3 nsecs
  – SunLab machines (Intel Core i5-4690) run at 3.5 GHz
• Basic operations take 1 – 10 clock cycles
  – .3 – 3 nsecs
• Accessing memory takes 70-100 nsecs
• How is this made to work?
Caching to the Rescue
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 computer system structure]

- CPU chip
- Register file
- ALU
- Cache memories
- Bus interface
- System bus
- Memory bus
- I/O bridge
- Main memory
- I/O bridge
- Bus interface
- Cache memories
- ALU
- Register file
- CPU chip
General Cache Concepts

Cache

Memory

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”.

Data is copied in block-sized transfer units.

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

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

Data is copied in block-sized transfer units.
General Cache Concepts: Hit

Request: 14

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

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)
General Caching Concepts: Types of Cache Misses

• **Cold (compulsory) miss**
  – cold misses occur because the cache is empty

• **Conflict miss**
  – most caches limit blocks to a small subset (sometimes a singleton) of the block positions in RAM
    » e.g., block \(i\) in RAM must be placed in block \((i \mod 4)\) in the cache
  – conflict misses occur when the cache is large enough, but multiple data objects all map to the same cache block
    » e.g., referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

• **Capacity miss**
  – occurs when the set of active cache blocks (**working set**) is larger than the cache
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

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

Address of word:
- t bits
- s bits
- b bits

Data begins at this offset

B = 2^b bytes per cache block (the data)
Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes

\[
S = 2^s \text{ sets}
\]

Address of int:
\[
t \text{ bits } 0...01 \quad 100
\]
Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes

Address of int:

valid? + match: assume yes = hit

block offset

v
tag

0 1 2 3 4 5 6 7

t bits

0...01 100
Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes

Address of int:

valid? + match: assume yes = hit

Block offset

int (4 Bytes) is here

No match: old line is evicted and replaced
Direct-Mapped Cache Simulation

- \( t = 1 \), \( s = 2 \), \( b = 1 \)

M=16 byte addresses, B=2 bytes/block, 
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

\[
\begin{array}{ccc}
0 & [0000_2], & \text{miss} \\
1 & [0001_2], & \text{hit} \\
7 & [0111_2], & \text{miss} \\
8 & [1000_2], & \text{miss} \\
0 & [0000_2] & \text{miss}
\end{array}
\]

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1 0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1 0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j
assume: cold (empty) cache, a[0][0] goes here
32 B = 4 doubles
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; i++)
        for (i = 0; i < 16; j++)
            sum += a[i][j];

    return sum;
}
```
Conflict Misses: Aligned

double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}
Different Alignments

def dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}
E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set
Assume: cache block size 8 bytes

Address of short int:

0...01 100

compare both

valid? + match: yes = hit

block offset

v  tag  0 1 2 3 4 5 6 7

v  tag  0 1 2 3 4 5 6 7
E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set
Assume: cache block size 8 bytes

Address of short int:
\[ t \text{ bits} \quad 0...01 \quad 100 \]

valid? + match: yes = hit

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...
Given the address above and the cache contents as shown, what is the value of the int at the given address?

a) 1111  
b) 3333  
c) 4444  
d) 7777
2-Way Set-Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Block</th>
<th>Hit/miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>hit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles
A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```
Conflict Misses

double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}
Intel Core i5 and i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

- ... -

L1 i-cache and d-cache:
- 32 KB, 8-way,
- Access: 4 cycles

L2 unified cache:
- 256 KB, 8-way,
- Access: 11 cycles

L3 unified cache:
- 8 MB, 16-way,
- Access: 30-40 cycles

Block size: 64 bytes for all caches
What About Writes?

• Multiple copies of data exist:
  – L1, L2, main memory, disk

• What to do on a write-hit?
  – write-through (write immediately to memory)
  – write-back (defer write to memory until replacement of line)
    » need a dirty bit (line different from memory or not)

• What to do on a write-miss?
  – write-allocate (load into cache, update line in cache)
    » good if more writes to the location follow
  – no-write-allocate (writes immediately to memory)

• Typical
  – write-through + no-write-allocate
  – write-back + write-allocate