Memory

• Memory technologies
• Memory hierarchy
  – Cache basics
  – Cache variations
  – Virtual memory
• Synchronization

Memory Technologies

• Read Only Memory (ROM)
• Static RAM (SRAM)
  – Basic cell: clocked D latch (vs. D flip flop)
  – Array of cells: 1-dimension and 2-dimension
  – RAM access: timing
• Dynamic RAM (DRAM)
  – Basic cell
  – Faster DRAM, e.g. synchronous DRAM
ROM

- *Random access* means that the access time (to read or write) is not dependent on the address -- different from hard disks
- ROM is random access but *read only*
- Programmable ROM (PROM)
- Erasable PROM (EPROM)
- Electrically EPROM (EEPROM)

RAM

- RAM is read and *writeable* random access memory
- Memory is erased when power goes off
  - ROM doesn’t lose memory when power goes off
- Static RAM: faster but more expensive
- Dynamic RAM: slower, smaller, and less expensive but it’s getting faster
Components: CMOS

- **n-channel transistor**
  - Source
  - Gate
  - Drain
  - gate = ’1’ → close
  - gate = ’0’ → open

- **p-channel transistor**
  - Source
  - Gate
  - Drain
  - gate = ’1’ → open
  - gate = ’0’ → close

Switch and NOR

Switch:
- When C = 1 then switch is closed, i.e., A = B.
- When C = 0, switch is open
Simple Memory Cell: D Latch

Similar to a D flip flop.
Clock = L, it holds its value.
Clock = H, Q = D,
but it's transparent from input to output.

D Flip Flop

Master Slave

Clock = L
Hold Transparent

Clock = H
Hold Transparent

Last value of D while Clock = L

Galen Sasaki  EE 361 University of Hawaii

7
Static RAM

Common Type

Chip Select (CS):
asserting it will turn on the RAM

R/W:
asserting it, will allow you to read it
unasserting it, will allow you to write to it

Address should be stable while CS = 1

1-Bit SRAM Cell

Let's build it in steps.
First consider just R/W

Add in CS
Dynamic RAM

- Memory cells can forget!
- Cells must be refreshed
- Less number of devices per bit
- Read destroys bits
  - Write bit back after a read

A cell

<table>
<thead>
<tr>
<th>Word line</th>
<th>Word line</th>
<th>Word line</th>
<th>Word line</th>
<th>Word line</th>
<th>Word line</th>
<th>Word line</th>
<th>Word line</th>
<th>Sense/write amplifiers -- sense and amplifier data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacitor</td>
<td>Capacitor</td>
<td>Capacitor</td>
<td>Capacitor</td>
<td>Capacitor</td>
<td>Capacitor</td>
<td>Capacitor</td>
<td>Capacitor</td>
<td>CS</td>
</tr>
<tr>
<td>Bit line</td>
<td>Bit line</td>
<td>Bit line</td>
<td>Bit line</td>
<td>Bit line</td>
<td>Bit line</td>
<td>Bit line</td>
<td>Bit line</td>
<td>data</td>
</tr>
</tbody>
</table>

4M x 1 DRAM

- Row decoder
  - 11-to-2048

- 2048 x 2048 array

- Column latches

- Mux

- Dout

- Latch 2048 bits at a time
- Throw away 2047
Memory Hierarchy

• Motivation, i.e., Why?
• Cache basics
  – Contents
  – Reading
    • Replacement policy
  – Writing
    • Data consistency
• Example: Direct Mapped
• Blocks of memory: better efficiency

Memory Hierarchy

• Making large fast memory cheaply.

<table>
<thead>
<tr>
<th>Memory Types</th>
<th>Access Times</th>
<th>Cost/Mbyte (93)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static RAM (SRAM)</td>
<td>fast (8-35 ns)</td>
<td>$100 - 400</td>
</tr>
<tr>
<td>Dynamic RAM (DRAM)</td>
<td>slower (90 - 120 ns)</td>
<td>$25-50</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>even slower (10-20 M ns)</td>
<td>$1-2</td>
</tr>
</tbody>
</table>
Cache: Why does it work?

Principle of locality: observation about executing programs

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced soon, e.g., loops
- Spatial locality (locality in space): if an item is referenced, items that have nearly the same address values will tend to be referenced soon.

Temporal Locality

main: add $1,$2,$3  spatial locality
and $2,$5,19

loop: beq skip
temporal locality

skip:  j loop

Galen Sasaki EE 361 University of Hawaii 17
Cache Basics

- Memory access: read/write
  - hit: find element in cache
  - miss: element not in cache

- Replacement policy: if we store something new in the cache, where do we write it?
  - Example: Least Recently Used (LRU) -- the "oldest" one.

Cache Contents

- valid bit: used or unused
- tag: main memory address--it's an identifier of where the data comes from
- age: used by LRU policy--this is updated every time the memory address is accessed
**Read**

**Step 1**
- Processor requests addr 1024
- Check if 1024 is in cache (check tags)

**Step 2**
- Processor loads 1024 in cache and (possibly) replaces something.
- Update age of 1024

**Write**

**Step 1**
- Processor writes addr 1024
- Check if 1024 is in cache (check tags)

**Step 2**
- Processor updates age of 1024
- Write addr 1024

? Depends on write policy
- Depends on write policy

HIT
- HIT

MISS
- MISS

Replace something
- Replace something

Replace something
- Replace something
Memory Write

Three approaches:

1. **Write-through**: write to both cache and main memory. Simple, but large overhead.

   ![Write-through diagram]

2. **Write-buffer**: write-through, but there is a buffer that takes care of the writing to memory.

   Buffer allows CPU to go onto the next operation.

   If buffer is not finished before the next memory access then CPU stalls until the write-through is finished.

   ![Write-buffer diagram]

3. **Write-back**: write to cache

   Update main memory when item in cache is replaced. More complicated.

   ![Write-back diagram]

   "dirty bit": indicates if the contents have to be written back to memory if replaced in cache.
Example: Direct Mapped

- Caching: Variations

1. Where do you place a new entry? Or which do you replace?
   E.g., Least Recently Used
2. In case of write, when do you write back into main memory?
   E.g., write-through (always)

Direct Mapped deals with 1.

You want to place the contents of memory location A into the cache.

You use the value of A to determine this.

Thus, A is always placed into the same position in the cache.

(Different than LRU)

Most common way is to use the last few bits of A.

Easy to find data in cache (don’t have to search entire contents)

Direct Mapped Example

Use the last two bits to determine the cache index to map memory contents to.

Why the last two bits, rather than the first two bits?
Direct Mapped Example

Cache Contents

valid bit:
used
unused
tag
main memory address minus last bits
main memory address = tag + cache index (e.g., 01)
Direct Mapped Hardware

- CPU
- address
- byte offset (2 bits)
- Cache
- index
- Hit
- data
- tag

Blocks of Memory

- Speed things up by organizing memory into "blocks" of words.
  - Access "blocks" instead of words
  - Take advantage of spatial locality of references

- Address
- Block Number or Block Address
- Block Offset (indicates where the word is in the block)

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
**Memory Organization**

- Main memory
- Interleaved

- 4-word block
- Cache
- CPU

* Access time of block is approx. access time of a single word.

**Cache**

- Address
- Block #
- Block offset
- Tag index
- Bit offset

- Tag
- Mux
- Hit
- Data

Galen Sasaki  
EE 361 University of Hawaii
Memory Hierarchy

Bigger blocks mean smaller miss rates

BUT

Higher miss penalties

TRADE OFF

Designing memory systems to support caches

Example:
1 clock cycle to send address
15 clock cycles for each DRAM access initiated
1 clock cycle to send a word of data
Memory Access

READ: same as before

Step 1. Check if word is in cache
Step 2.
  HIT: Read word from cache
  MISS: Read block (and word) from main memory.
  Put block in cache

WRITE: different for write through

Step 1.
  Read block from main memory into cache
  Write into main memory and cache

Cache Variations

- Associative memory
- Set associate memory
- Two levels of caching
Associative Memory

- **Cache Contents**
  - Each element contains: tag, data, valid bit, age

- **Access Cache**
  - Check cache by checking all elements (check tags of all elements to see if there's a hit)
  - Replacement policy: can be more intelligent to reduce misses
    - E.g., least recently used (LRU): need to update age field
    - E.g., random

- Lowers miss rate but increases hardware complexity (and maybe speed)
Set Associative

* **Associative:** Reduces the chance of a miss. Large complexity because all blocks must be searched.

* **Direct-Mapped:** Simpler complexity

* **Set Associative** (Generalization of above):

  A block is mapped to a set just as in *direct mapped*

  A block may be mapped anywhere within its set just as in *associative*

  n-way set associative means n blocks in a set

---

Set Associative

Main Memory

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
<th>Block 4</th>
<th>Block 5</th>
<th>Block 6</th>
<th>Block 7</th>
</tr>
</thead>
</table>

A block is a “super word”
Block #s are the addresses

Word address 00

block size = $2^k$ words

$m$ bits

$2^m$ sets

set index

n-way set associative cache

associative with n blocks

direct map to the set using set index
Cache Performance

A. CPU time = ( CPU execution clock cycles + memory-stall clock cycles) x clock cycle time.

memory-stall clock cycles = read-stall cycles + write-stall cycles

B. Read-stall cycles = reads/program * read miss rate * read miss penalty

C. Write-stall cycles = writes/program * write miss rate * write miss penalty + write buffer stalls

usually negligible

D. Simplifying case: Write and read penalties are about the same, and write buffer stalls are negligible.

Memory-stall cycles = memory access/program * miss rate * miss penalty

Memory Hierarchy

- Miss rate vs. Associativity for different cache sizes (1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB)
- Bars represent different associativity levels: One-way, Two-way, Four-way, Eight-way

Galen Sasaki EE 361 University of Hawaii
Improving Performance

Set associative memory
Choose set and block size so that
• hardware is fast (hit time)
• minimize miss rate

CPU

cache

Main memory

Two Level Cache

Min hit time

Min miss rate

Lowers miss penalty of L1 cache

Level 1 cache

Level 2 cache

Main memory

Virtual Memory

• Really big memory
  – E.g., there are 32 bits in an address, addressing up to around 4 billion bytes. 64 bit addresses address much much more. Much bigger than main memory.
• Virtual memory is this really big memory but data is physically stored in all kinds of stuff
Virtual Memory

Really big memory

“Pages” instead of “blocks”

Pages are reasonably big chunks of bytes

Pages are stored in main memory (physical memory) or hard disk

How do you keep track of where the pages are?

Virtual Memory

Main memory

Page Table

Processor page table register

Page Table does address translation

Main memory

“a cache for hard disk”

Hard disk

Etc

Pages instead of blocks

Pages are reasonably big chunks of bytes

Pages are stored in main memory (physical memory) or hard disk

Address Translation

virtual addresses

5

7

Page 0

Page 1

Page 2

Page 3

Page 4

Page 5

Page 6

Page 7
Accessing Virtual Memory

Virtual address

<table>
<thead>
<tr>
<th>virtual page #</th>
<th>page offset</th>
</tr>
</thead>
</table>

Translation lookaside buffer (TLB)  
(It’s a cache for the page table)

Page table

Use offset to get the exact word or byte within the page

Virtual Memory

- Disk accesses are slow
  - memory management can be done in software
  - page sizes are large (10s of Kbytes) to mitigate slow access times of disks
  - miss penalties are very large so
    - associative memory
    - least recently used or even more complicated strategies
    - write-back rather than write-through
Virtual Memory

• Programs do not have to be in contiguous portions of memory. They can be on pages located anywhere.

• Note that strategy for virtual memory is different than for caches because of the high miss penalties