CS61C : Machine Structures

Lecture 6.1.2
Cache I

2004-07-27

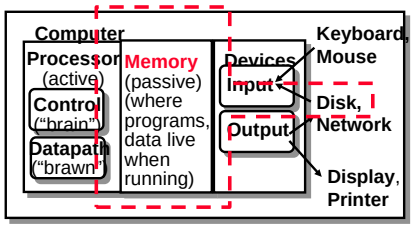
Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

Design Principles for Hardware

1. **Simplicity favors regularity**
   - Every instruction has 3 operands, opcode same place for EVERY instr
2. **Smaller is faster**
   - 32 registers, no more
3. **Good design demands good compromise**
   - MIPS Immediate format vs. R format
4. **Make the common case fast**
   - Support of constants via immediates

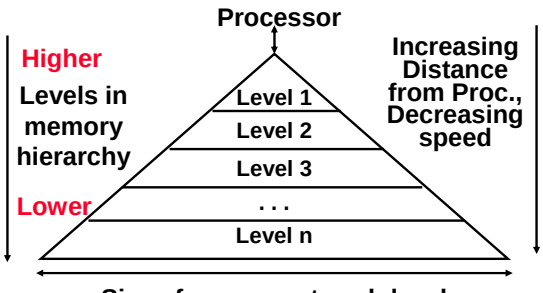
The Big Picture



Memory Hierarchy (1/3)

- **Processor**
  - executes instructions on order of nanoseconds to picoseconds
  - holds a small amount of code and data in registers
- **Memory**
  - More capacity than registers, still limited
  - Access time ~50-100 ns
- **Disk**
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds

Memory Hierarchy (2/3)



As we move to deeper levels the latency goes up and price per bit goes down.

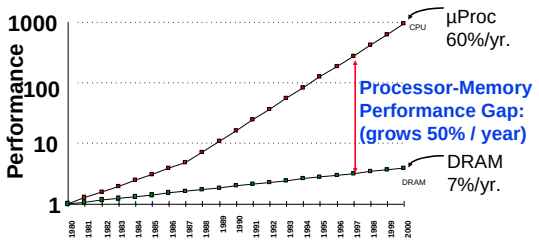
Memory Hierarchy (3/3)

- **If level closer to Processor, it must be:**
  - smaller
  - faster
  - subset of lower levels (contains most recently used data)
- **Lowest Level (usually disk) contains all available data**
- **Other levels?**

Trend Comparison

- Processor Performance
  - Cycle time – Exponential Growth
  - # Transistors – Exponential Growth
- Memory Performance
  - RAM size – Exponential
  - Disk Size – Exponential
  - RAM/Disk Access Time – Roughly Linear/Slow exponential

Trend Comparison



- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two levels of cache on chip

Memory Caching

- We've discussed three levels in the hierarchy: processor, memory, disk
- Mismatch between processor and memory speeds leads us to add a new level: a memory cache
- Implemented with SRAM technology: faster but more expensive than DRAM memory.

Memory Hierarchy Analogy: Library (1/2)

- You're writing a term paper (Processor) at a table in Doe
- Doe Library is equivalent to disk
  - essentially limitless capacity
  - very slow to retrieve a book
- Table is memory
  - smaller capacity: means you must return book when table fills up
  - easier and faster to find a book there once you've already retrieved it

Memory Hierarchy Analogy: Library (2/2)

- Open books on table are cache
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library

Memory Hierarchy Analogy: Library (3/2)

- At any 1 time, you have a working set of books you access most often…
  - Your working set shifts over time (maybe study Schopenhauer then Nietzsche).
  - But it's always a small subset of the books in the library.
  - As long as you can keep your working set close to you, your "*book effective access time*" is very fast
- Programs work just the same way …

Memory Hierarchy Basics

- Programs exhibit "temporal locality" …
  - If we just accessed it, chances are we'll access it again soon.

    Or, more formally,
  - The probability of accessing a particular piece of data varies inversely with the time since we last accessed it.
  - FYI: Working set is theoretical. Temporal locality is a way of finding the working set of a program.

Memory Hierarchy Basics

- Disk contains everything.
- Memory contains copies of data on disk that are being used.
- Cache contains copies of data in memory that are being used.
- When Processor needs something, bring it into to all higher levels of memory.

Cache Design

- Big Four Questions of Cache Design
  - Where does each memory address map to?

    (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
  - How do we know which elements are in cache?
  - How do we quickly locate them?
  - How/when do we replace things

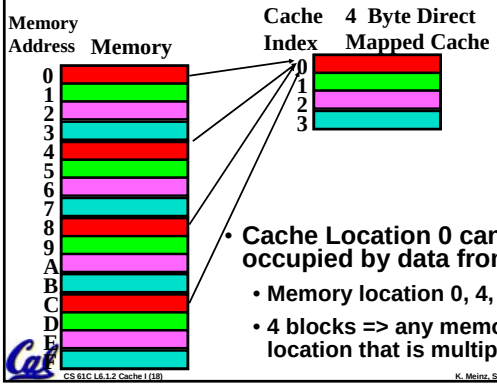
Cache Design: Four Questions

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Direct-Mapped Cache (1/2)

- In a direct-mapped cache, each memory address is associated with one possible block within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

Direct-Mapped Cache (2/2)



- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - 4 blocks => any memory location that is multiple of 4

Issues with Direct-Mapped

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Result: divide memory address into three fields

tag	index	byte
to check	to	offset
if have	select	within
correct block	block	block

Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- Index: specifies the cache index (which "row" of the cache we should look in)
- Offset: once we've found correct block, specifies which byte within the block we want
- Tag: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

Caching Terminology

- When we try to read memory, 3 things can happen:

1. cache hit: cache block is valid and contains proper address, so read desired word
2. cache miss: nothing in cache in appropriate block, so fetch from memory
3. cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Direct-Mapped Cache Example (1/3)

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
  - need to specify correct byte within a block
  - block contains 4 words
    = 16 bytes
    = 2^4 bytes
  - need 4 bits to specify correct byte

Direct-Mapped Cache Example (2/3)

- Index: (~index into an "array of blocks")
  - need to specify correct row in cache
  - cache contains 16 KB = 2^{14} bytes
  - block contains 2^4 bytes (4 words)
  - # blocks/cache
    = $\frac{\text{bytes/cache}}{\text{bytes/block}}$
    = $\frac{2^{14}\text{ bytes/cache}}{2^4\text{ bytes/block}}$
    = 2^{10} blocks/cache
  - need 10 bits to specify this many rows

Direct-Mapped Cache Example (3/3)

- Tag: use remaining bits as tag
  - tag length = addr length – offset - index
    = 32 - 4 - 10 bits
    = 18 bits
  - so tag is leftmost 18 bits of memory address
- Why not full 32 bit address as tag?
  - All bytes within block need same address (4b)
  - Index must be same for every address within a block, so its redundant in tag check, thus can leave off to save memory (10 bits in this example)

Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
- Read 4 addresses
  1. 0x00000014
  2. 0x0000001C
  3. 0x00000034
  4. 0x00008014
- Memory values on right:
  - only cache/ memory level of hierarchy

Memory Address (hex)	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

Accessing data in a direct mapped cache

- 4 Addresses:
  - 0x00000014, 0x0000001C, 0x00000034, 0x00008014
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
00000000000000000000 0000000001 0100
00000000000000000000 0000000001 1100
00000000000000000000 0000000011 0100
00000000000000000010 0000000001 0100
```

Tag Index Offset

16 KB Direct Mapped Cache, 16B blocks

- Valid bit: determines whether anything is stored in that row (when computer initially turned on, all entries are invalid)

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

1. Read 0x00000014

- 00000000000000000000 0000000001 0100
  Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So we read block 1 (0000000001)

- 00000000000000000000 0000000001 0100
  Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

No valid data

- 00000000000000000000 0000000001 0100
  Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read from cache at offset, return word b

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

2. Read 0x0000001C = 0…00 0..001 1100

- 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index is Valid

- 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index valid, Tag Matches

- 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index Valid, Tag Matches, return d

- 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

3. Read 0x00000034 = 0…00 0..011 0100

- 00000000000000000 0000000011 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

So read block 3

- 00000000000000000 0000000011 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

No valid data

- 00000000000000000 0000000011 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Load that cache block, return word f

- 00000000000000000 0000000011 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

4. Read 0x00008014 = 0…10 0..001 0100

- 00000000000000010 0000000001 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

- 00000000000000010 0000000001 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Cache Block 1 Tag does not match (0 != 2)

- 00000000000000010 0000000001 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

- 00000000000000010 0000000001 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

And return word j

- 00000000000000010 0000000001 0100
  Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a ,b, c, d, e, ..., k, l
- Read address 0x00000030 ?
  00000000000000000 0000000011 0000
- Read address 0x0000001c ?
  00000000000000000 0000000001 1100

Cache

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		

Answers

- 0x00000030 a hit
  Index = 3, Tag matches, Offset = 0, value = e
- 0x0000001c a miss
  Index = 1, Tag mismatch, so replace from memory, Offset = 0xc, value = d
- Since reads, values must = memory values whether or not cached:
  - 0x00000030 = e
  - 0x0000001c = d

Memory

Address	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001c	d
...	...
00000030	e
00000034	f
00000038	g
0000003c	h
...	...
00008010	i
00008014	j
00008018	k
0000801c	l
...	

And in Conclusion...

- Mechanism for transparent movement of data among levels of a storage hierarchy
  - set of address/value bindings
  - address ⇒ index to set of candidates
  - compare desired address with tag
  - service hit or miss
    - load new block and binding on miss

address: tag index offset
00000000000000000 0000000001 1100

Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0					
1	0	a	b	c	d
2					
3					