On Monday the Library of Congress added 5 exceptions to the DMCA that, among other things, verify the legality of jailbreaking devices such as phones (details @ the URL). This means you can actually hack around with devices you already own. Ground breaking, isn’t it?

Instructor Paul Pearce

Review : Pipelining

- Pipeline challenge is hazards
  - Forwarding helps with many data hazards
  - Delayed branch helps with control hazard in our 5 stage pipeline
  - Data hazards w_Loads → Load Delay Slot
    - Interlock → “smart” CPU has HW to detect if conflict with inst following load, if so it stalls
- More aggressive performance (discussed in section a bit today)
  - Superscalar (parallelism)
  - Out-of-order execution

Motivation: Why We Use Caches (written $)

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

Memory Hierarchy ⎯ i.e., storage in computer systems

- Processor
  - Holds data in register file (~100 Bytes)
  - Registers accessed on nanosecond timescale
- Memory (we’ll call “main memory”)
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access?!?
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds

Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level: a memory cache
- Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory.
- Most processors have separate caches for instructions and data. This is how we solved the single-memory structural hazard yesterday.
Memory Hierarchy

Processor

Increasing Distance from Proc., Decreasing speed

Level n

Level 3

Level 2

Level 1

Higher Levels in memory hierarchy

Lower Levels in memory hierarchy

Size of memory at each level
As we move to deeper levels the latency goes up and price per bit goes down.

Memory Hierarchy

• If level closer to Processor, it is:
  • Smaller
  • Faster
  • More expensive
  • Subset of lower levels (contains most recently used data)

• Lowest Level (usually disk) contains all available data (does it go beyond the disk? Is it networked? The cloud?)

• Memory Hierarchy presents the processor with the illusion of a very large & fast memory

Memory Hierarchy Analogy: Library (1/2)

• You're writing a term paper (Processor) at a table in your dorm

• Doe Library is equivalent to disk
  • Essentially limitless capacity
  • Very slow to retrieve a book

• Dorm room is main memory
  • Smaller capacity: means you must return book when dorm room 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, put it away (in your dorm)
  • 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 in your dorm as possible, since faster than going to library

• In reality, disk is SO slow, its more like having to drive to the Stanford Library

• And who wants to do that?

Memory Hierarchy Basis

• Cache contains copies of data in memory that are being used.

• Memory contains copies of data on disk that are being used.

• Caches work on the principles of temporal and spatial locality.
  • Temporal Locality: if we use it now, chances are we'll want to use it again soon.
  • Spatial Locality: if we use a piece of memory, chances are we'll use the neighboring pieces soon.

Cache Design

• How do we organize cache?

• 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?
Administrivia

- Homework 8 due tonight at midnight
- Project 2: Find a partner, get going. Due Monday.
- Project 1 and HW4 grades up now.
  - We had to regrade the project a few times to make sure the distribution was fair, and it took longer than expected. Sorry.
  - HW5 and 6 are in the pipeline. Should be done soon.
- Reminder: The drop deadline is this FRIDAY. I believe you have to drop in person, so don’t wait.

Direct-Mapped Cache (1/4)

- 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 (2/4)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Index</th>
<th>4 Byte Direct Mapped Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Block size = 1 byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cache Location 0 can be occupied by data from:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Memory location 0, 4, 8, ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 4 blocks ⇒ any memory location that is multiple of 4</td>
</tr>
</tbody>
</table>

What if we wanted a block to be bigger than one byte?

Memory Address

0 1 2 3 4 5 6 7 8 9 A B C D E

Direct-Mapped Cache (3/4)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Index</th>
<th>8 Byte Direct Mapped Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Block size = 2 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- When we ask for a byte, the system finds out the right block, and loads it all!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- How does it know right block?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- How do we select the byte?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- E.g., Mem address 11101?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- How does it know WHICH colored block it originated from?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- What do you do at baggage claim?</td>
</tr>
</tbody>
</table>

Direct-Mapped Cache (4/4)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Index</th>
<th>8 Byte Direct Mapped Cache w/Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Tag (Block size = 2 bytes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- What should go in the tag?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Do we need the entire address?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- What do all these tags have in common?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- What did we do with the immediate when we were branch addressing, always count by bytes?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Why not count by cache #?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- It’s useful to draw memory with the same width as the block size</td>
</tr>
</tbody>
</table>

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?
- Answer: divide memory address into three fields

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Address</th>
<th>Cache Index</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- Index
  - specifies the cache index (which “row”/block 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

Direct-Mapped Cache Example (1/3)

- Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
  - Sound familiar?
- 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 2 bytes = 2^1 bytes
    - need 1 bit to specify correct byte

Direct-Mapped Cache Example (2/3)

- Index: (~index into an “array of blocks”)  
  - need to specify correct block in cache  
  - cache contains 8 B = 2^3 bytes  
  - block contains 2 B = 2^1 bytes  
  - # blocks/cache  
    - \[ \text{bytes/cache} \]  
    - \[ \text{bytes/block} \]  
    - \[ \text{2^1 bytes/block} \]  
    - \[ \text{2^2 blocks/cache} \]  
  - need 2 bits to specify this many blocks

Direct-Mapped Cache Example (3/3)

- Tag: use remaining bits as tag
  - tag length = addr length – offset - index  
    - 32 - 1 - 2 bits  
    - 29 bits
  - so tag is leftmost 29 bits of memory address
- Why not full 32 bit address as tag?
  - Not all bytes within a block have the same address. So we shouldn’t include offset  
  - Index is the same for every address within a block, so it’s redundant in tag check, thus can leave off to save memory (here 8 bits)

Caching Terminology

- When reading memory, 3 things can happen:
  - cache hit: cache block is valid and contains proper address, so read desired word
  - cache miss: nothing in cache in appropriate block, so fetch from memory
  - cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)
Accessing data in a direct mapped cache

Ex.: 16KB of data, direct-mapped, 4 word blocks

- Can you work out height, width, area?

- Read 4 addresses
  1. 0x00000014
  2. 0x0000001C
  3. 0x00000034
  4. 0x00008014

Memory vals here:

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Value of Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000010</td>
<td>a</td>
</tr>
<tr>
<td>00000014</td>
<td>b</td>
</tr>
<tr>
<td>00000018</td>
<td>c</td>
</tr>
<tr>
<td>0000001C</td>
<td>d</td>
</tr>
<tr>
<td>00008010</td>
<td>i</td>
</tr>
<tr>
<td>00008014</td>
<td>j</td>
</tr>
<tr>
<td>00008018</td>
<td>k</td>
</tr>
</tbody>
</table>

Direct-Mapped Cache Example (2/3)

- Offset
  - block contains 16 bytes
    = 2^4 bytes \rightarrow 4 bits for offset

- Index:
  - cache contains 16 KB = 2^{14} bytes
    = 2^{14} bytes/cache
    = 2^2 bytes/block
    = 2^{10} blocks/cache \rightarrow 10 bits for index

- Tag:
  - 32 bit address \rightarrow 10 bits for index, 4 for offset
    = 32 – 10 – 4 = 18 bits for tag

16 KB Direct Mapped Cache, 16B blocks

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>0xc-f</th>
<th>0x8-b</th>
<th>0x4-7</th>
<th>0x0-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>3</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>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1022</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1023</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Load word from 0x00000014

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>1</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>2</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>3</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>4</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>5</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>6</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>7</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1022</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1023</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

So we read block 1 (0x00000001)

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>1</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>2</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>3</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>4</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>5</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>6</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>7</td>
<td>0x-c-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1022</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1023</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
No valid data

- 0000000000000000 000000001 0100
  
  Valid
  
  Index  Tag  0xc-f  0x8-b  0x4-7  0x0-3
  0  1  d  c  b  a
  1  2  3  4  5  6
  6  7

...  ...

1022  ...
1023  ...

Valid

- 0000000000000000 000000001 0100
  
  Tag field  Index field  Offset
  0  1  2  3  4  5  6  7

...  ...

1022  ...
1023  ...

So load that data into cache, setting tag, valid

- 0000000000000000 000000001 0100
  
  Valid
  
  Index  Tag  0xc-f  0x8-b  0x4-7  0x0-3
  0  1  d  c  b  a
  1  2  3  4  5  6
  6  7

...  ...

1022  ...
1023  ...

Read from cache at offset, return word b

- 0000000000000000 000000001 0100
  
  Valid
  
  Index  Tag  0xc-f  0x8-b  0x4-7  0x0-3
  0  1  d  c  b  a
  1  2  3  4  5  6
  6  7

...  ...

1022  ...
1023  ...

2. Load word from 0x0000001C

- 0000000000000000 000000001 1100
  
  Valid
  
  Index  Tag  0xc-f  0x8-b  0x4-7  0x0-3
  0  1  d  c  b  a
  1  2  3  4  5  6
  6  7

...  ...

1022  ...
1023  ...

Index is Valid

- 0000000000000000 000000001 1100
  
  Valid
  
  Index  Tag  0xc-f  0x8-b  0x4-7  0x0-3
  0  1  d  c  b  a
  1  2  3  4  5  6
  6  7

...  ...

1022  ...
1023  ...

Index valid, Tag Matches
### Index Valid, Tag Matches, return d

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xC-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0-3</td>
<td></td>
</tr>
</tbody>
</table>

### Peer Instruction

1. All caches take advantage of spatial locality.
2. All caches take advantage of temporal locality.
3. If you know your computer’s cache size, you can often make your code run faster.

### And in Conclusion…

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
  - each successively lower level contains “most used” data from next higher level
  - exploits temporal & spatial locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea