COMPUTER ARCHITECTURE

MEMORY

Dr. Bill Yi
Santa Clara University

(Also based on presentation: Dr. Nam Ling, COEN210 Lecture Notes)
COURSE CONTENTS

- Introduction
- Instructions
- Computer Arithmetic
- Performance
- Processor: Datapth
- Processor: Control
- Pipelining Techniques
- Memory
- Input/Output Devices
MEMORY

- Memory Hierarchy
- Cache: Basics, Mapping
- Cache: Performance, Associativity
- Memory Technology & Interleaving
- Multilevel Caches
- Virtual Memory: Basics, Page Tables & TLBS
- Integrating Virtual Memory, TLBs, Caches
Memories: Review

- **SRAM:**
  - Value is stored on a pair of inverting gates
  - Very fast but takes up more space than DRAM

- **DRAM:**
  - Value is stored as a charge on capacitor (must be refreshed)
  - Very small but slower than SRAM
Exploiting Memory Hierarchy

- Processor - DRAM performance gap growing; users want large and fast memories!

  **SRAM** access times are 0.5 – 5 ns at cost of $4,000 to $10,000 per GB.
  **DRAM** access times are 50 – 70 ns at cost of $100 to $200 per GB.
  **Magnetic Disk** access times are 5 to 20 million ns at cost of $0.50 to $2 per GB.
  (2004 data)

- Build a memory hierarchy
- Goal of memory hierarchy: to achieve
  - speed of highest level (cache)
  - size of lowest level (disk)
Exploiting Memory Hierarchy

- A typical memory hierarchical system

- CPU
  - Register/Register file

- L1 Cache
  - SRAM

- L2 Cache
  - SRAM

- Main (Primary) Memory
  - DRAM

- Hard Disk (Secondary Memory)
  - Magnetic Disk

On-chip
Locality

- A principle that makes having a memory hierarchy a good idea

- If an item is referenced,
  **temporal locality:** it will tend to be referenced again soon
  **spatial locality:** nearby items will tend to be referenced soon.

- Memory hierarchy:
  - keep recently accessed data closer to processor --- temporal
  - move blocks consisting of contiguous words to upper level --- spatial
  
  *Why does code have locality?*

- Our initial focus: two levels (upper, lower)
  - block: minimum unit of data to transfer between memory levels
  - hit: data requested is in the upper level
  - miss: data requested is not in the upper level
  - Hit rate = #hit/#access
  - Miss rate = 1 - hit rate
  - Miss penalty: time (transfer the block to upper level) + time (delivery data to processor)
**Cache**

- **Cache**: the level of memory between CPU and main memory; typically on-chip SRAM or off-chip SRAM of KB-MB size (main memory is typically DRAM of GB size)
- Caches first appeared in research computers in early 60s and in production computers later in same decade – all general-purpose computers today have caches
- Cache takes advantage of locality of access (try to contain the active portion of a program)
- Two issues:
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?
Our first example: block size is one word of data

Direct Mapping: \((\text{Block address}) \mod (\text{Number of blocks in cache})\)
- Ex. Block 21 maps to Cache block 21 Mod 8 = Cache block 5

For each item of data at the lower level, there is exactly one location in the cache where it might be.
E.g. lots of items at the lower level share locations in the upper level
Direct Mapping: Addressing

- **Index**: (Block address) modulo (Number of blocks in cache), used to select a block in the cache.
- **Tag**: upper portion of address (bits that are not used as index into the cache), to identify if the block in the cache contains requested word; a comparator can be used to match.
- **Valid bit**: to indicate whether an entry contains a valid address (e.g. when CPU starts up, tag bits meaningless).
- **Block offset**: to identify word within selected block.
- **Byte offset**: to identify byte within selected word.

Recently accessed word in cache → **temporal locality**
When a miss occurs, we fetch a block of contiguous words from memory to cache → **spatial locality**

Basically, when a miss occurs, we stall the CPU, freeze the content of registers; a separate controller handles cache miss, fetches data/instructions from main memory to cache.
Direct Mapping: Addressing

- Accessing a word in a 64KB cache, with 4 words per block

Address (showing bit positions)

31 · · · 16 15 · · · 4 3 2 1 0

V Tag Data

Hit

Block offset

Data

Direct Mapping: Addressing

4K entries

Mux

32

16 32 32 32

Index

16 12 2 Byte offset

128 bits

16 bits

Tag

V

Data

Hit

Tag

16 bits
Direct Mapping: Example 1

Memory = 32 blocks = 64 words, cache = 4 blocks = 8 words, block = 2 words
Hence 6-bit word address: bit5-bit3 = tag, bit2-bit1 = index, bit0 = block offset

Word access sequence with word addresses (here we ignore byte addresses): 101100, 110100, 101100, 101101, 100001, 101100 (or memory word 44, 52, 44, 45, 33, 44)
Access sequence (block-word): 22-0, 26-0, 22-0, 22-1, 16-1, 22-0
Note: For each miss, 1 block (2 words) is transferred from memory to cache

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>Word0</th>
<th>Word1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1st access: miss, transfer block 22

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>Word0</th>
<th>Word1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2nd access: miss, transfer block 26, replace block 22

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>Word0</th>
<th>Word1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>110</td>
<td>Mem[52]</td>
<td>Mem[53]</td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3rd access: miss, transfer block 22, replace block 26

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>Word0</th>
<th>Word1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Y</td>
<td>100</td>
<td>Mem[32]</td>
<td>Mem[33]</td>
</tr>
<tr>
<td>01</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4th access: hit

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>Word0</th>
<th>Word1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Y</td>
<td>100</td>
<td>Mem[32]</td>
<td>Mem[33]</td>
</tr>
<tr>
<td>01</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5th access: miss, transfer block 16

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>Word0</th>
<th>Word1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Y</td>
<td>100</td>
<td>Mem[32]</td>
<td>Mem[33]</td>
</tr>
<tr>
<td>01</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6th access: hit
Cache = 64 blocks, block = 4 words (16 bytes), what block number does byte address 1200 map to?

Solution:
The block is given by (block address) mod (number of blocks in cache)
Address of block = Byte address / Bytes per block

Hence: Block number = ⌊1200/16⌋ mod 64
= 75 mod 64
= 11
Read hits
- this is what we want!

Read misses
- stall the CPU, fetch block from memory, deliver to cache, restart

Write hits:
- write data to both cache and memory *(write-through scheme)* (write buffer may be used to prevent lowering the performance too much)
- write the data only into the cache, the modified block is written to main memory when it must be replaced *(write-back scheme)*
- write-back improves performance but more complex to implement

Write misses:
- read the entire block into the cache, then write the word (using write-through or write-back)

Example: gcc on DECStation 3100, 64KB instruction cache, 64 KB data cache, 4 words per block
- instruction miss rate = 2.0%
- data miss rate = 1.7%
- effective combined miss rate = 1.9% (i.e. hit rate of 98.1%!)
Increasing the block size tends to decrease miss rate but increases miss penalty (transfer time)

Use separate caches for instructions & data because there is more spatial locality in code than in data
**Memory Interleaving to Support Caches**

- **Miss penalty** (transfer time incurred for a miss): Assume cache block = 4 words
  - Assume 1 cycle to send address to Memory, 15 cycles for each DRAM access initiated, 1 cycle to send a data word
  - Case 1. 1-word wide memory: miss penalty = 1 + 4 × 15 + 4 × 1 = 65 clock cycles
  - Case 2. Widening memory & bus to improve bandwidth (say 4 words): miss penalty = 1 + 1 × 15 + 1 × 1 = 17 clock cycles; however, high hardware cost
  - Case 3. Widening memory but not bus: Organize memory chips in banks to read/write multiple words in 1 access time; each bank is 1-word wide so width of bus & cache need not change (lower cost than Case 2). Sending an address to several banks permits them all to read simultaneously, hence retains the advantage of incurring full memory latency only once → **Memory Interleaving**
    - 4-way Interleaving: miss penalty = 1 + 1 × 15 + 4 × 1 = 20 clock cycles (very effective)
    - Interleaving: also valuable on writes, each bank can write independently, quadrupling write bandwidth

---

**Diagrams:**

- **a. One-word-wide memory organization**
- **b. Wide memory organization**
- **c. Interleaved memory organization**
## Memory Technology: Trends

<table>
<thead>
<tr>
<th>Year</th>
<th>DRAM Chip size</th>
<th>Access time (new row)</th>
<th>Column access time existing row</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64 Kbit</td>
<td>250 ns</td>
<td>150 ns</td>
</tr>
<tr>
<td>1983</td>
<td>256 Kbit</td>
<td>185 ns</td>
<td>100 ns</td>
</tr>
<tr>
<td>1985</td>
<td>1 Mbit</td>
<td>135 ns</td>
<td>40 ns</td>
</tr>
<tr>
<td>1989</td>
<td>4 Mbit</td>
<td>110 ns</td>
<td>40 ns</td>
</tr>
<tr>
<td>1992</td>
<td>16 Mbit</td>
<td>90 ns</td>
<td>30 ns</td>
</tr>
<tr>
<td>1996</td>
<td>64 Mbit</td>
<td>60 ns</td>
<td>12 ns</td>
</tr>
<tr>
<td>1998</td>
<td>128 Mbit</td>
<td>60 ns</td>
<td>10 ns</td>
</tr>
<tr>
<td>2000</td>
<td>256 Mbit</td>
<td>55 ns</td>
<td>7 ns</td>
</tr>
<tr>
<td>2004</td>
<td>512 Mbit</td>
<td>50 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>2006</td>
<td>512 Mbit</td>
<td>40 ns</td>
<td>2.5 ns</td>
</tr>
</tbody>
</table>

- Memory chips are organized to produce a number of output bits, usually 4 to 32 (with 16 being the most popular in 2006).
- DRAM: row access and column access (DRAMs buffer a row of bits inside for fast column access)
Recently – **DDR SDRAM** (double data rate synchronous DRAMs)

**SDRAM** (Synchronous DRAM): provides burst access to data from a series of sequential locations in DRAM. An SDRAM is supplied with a starting address & a burst length; data in burst transferred under control of a clock (e.g. up to 400 MHz in 2006).

Advantages of SDRAMs:
- Use of clock that eliminates the need to synchronize
- Elimination of the need to supply successive addresses in the burst

**DDR** (double data rate) – data transfers on both the leading and the falling edge of the clock (hence twice the bandwidth)

To deliver high bandwidth, internal DRAM is organized as **interleaved** memory banks
Simplified model:

**CPU time** = (CPU execution cycles + Memory-stall cycles) × cycle time

# of Memory-stall cycles = # of instructions × miss ratio × miss penalty

Two ways of improving performance:
- decreasing the miss ratio
- decreasing the miss penalty (multi-level caching)

**Example**: assume gcc program (I instructions, 36% of instructions are load/store). Icache miss rate = 2%, Dcache miss rate = 4%; machine CPI = 2 without memory stalls, miss penalty = 100 cycles. How much faster if never miss?

**Solution**: # of memory-stall cycles = I x 2% x 100 + I x 36% x 4% x 100 = 3.44I

CPU time with stalls = (2 + 3.44)I x clock cycle = 5.44I x clock cycle

CPU time with perfect cache = 2I x clock cycle

Ratio = CPI (stall) / CPI (perfect) = 5.44 / 2 = 2.72  (or 3.44/5.44 = 63%)

If we have faster CPU of CPI = 1, Ratio = 4.44 / 1 = 4.44  (or 3.44/4.44 = 77%!)

The importance of cache performance for CPUs with lower CPI & higher clock rates (see example in text) is greater!

**What happens if we increase block size?**
In **direct mapping**, a block is placed in exactly one location in cache.

In **fully associative mapping**, a block can be in **any** location in the cache.

In **n-way set associative mapping**, the cache is divided into **sets**, each consists of **n** blocks. Each block from memory maps into a unique set in cache given by index field, and the block can be placed in **any** block of that set.

All elements (tags) of the set must be searched to find a match for the block.

Compared to direct mapped, n-way set associative cache results in a lower miss ratio.

Increasing degree of associativity:
- **Advantage:** decrease miss rate
- **Disadvantage:** increase hit time (due to search for match)

Which entry in a set is to be replaced on a miss?

Typical solution: Replace the "**least recently used**" block within the set (LRU replacement strategy).
Example: 3 small caches, each 4 1-word blocks. Find number of misses for block sequence 0, 8, 0, 6, 8

- Direct mapped cache: 5 misses
- 2-way set associative: 4 misses
- Fully associative: 3 misses
4-way Set Associative Cache: Addressing

Compared to direct-mapped cache, set associative cache requires more hardware and even hit access time, but reduces miss rate.
**Performance: Effects of Associativity**

- **Example:** Improvement for SPEC2000 benchmarks for a 64 KB data cache with 16-word block:

- **Note in this case:** going from 1-way (direct mapped) to 2-way associativity – decreases miss rate by about 15%! But little improvement in going to higher associativity

- **Effects of cache size:** for smaller cache sizes, the improvement from associativity is higher – example:

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Data miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.3%</td>
</tr>
<tr>
<td>2</td>
<td>8.6%</td>
</tr>
<tr>
<td>4</td>
<td>8.3%</td>
</tr>
<tr>
<td>8</td>
<td>8.1%</td>
</tr>
</tbody>
</table>

![Graph showing miss rate vs cache size and associativity]
Multilevel Caches

Add a second level cache: Objective is to reduce miss penalty
- often primary cache (SRAM) is on the same chip as the processor
- use off-chip SRAM for a second level (L2) cache above primary memory (DRAM)
- miss penalty goes down if data is in L2 cache
Multilevel Caches: Performance

- **Effective CPI = CPU CPI + \( \sum \) (Memory-stall cycles per instruction)**

- **Example:** CPU CPI of 1.0 (e.g. pipelined) on a 5 GHz (0.2 ns cycle time) machine with a 2% miss rate at L1 cache, 100 ns DRAM access. How much faster if we add L2 cache with 5 ns access time which decreases miss rate to main memory to 0.5%?

- **Solution:**
  
  Miss penalty to main memory = 100 ns / (0.2 ns/cycle) = 500 clock cycles
  
  Effective CPI with just L1 cache = 1.0 + 2% x 500 = **11.0**
  
  Miss penalty for data found in L2 cache = 5 ns / (0.2 ns/cycle) = 25 clock cycles
  
  Effective CPI with L1 + L2 caches = CPU CPI + L1 stalls per inst. + L2 stalls per inst.
  
  = 1.0 + 2% x 25 + 0.5% x 500 = 1 + 0.5 + 2.5 = **4.0**
  
  Faster by **11.0 / 4.0 = 2.8**

- **Using multilevel caches:**
  - try to optimize the **hit time** on the **1st level cache**
  - try to optimize the **miss rate** on the **2nd level cache**
Virtual Memory (VM)

- **Virtual memory**: Main memory (DRAM) can act as a “cache” for the secondary storage (magnetic disk); a virtual memory block is called a **page**

- **Advantages**:
  - illusion of having more physical memory
  - program relocation (maps virtual addresses from program to physical addresses in memory)
  - protection

Note:
A collection of programs running at once on a machine, total memory required > main memory
Main memory needs to contain only active portions of the many programs
Programs sharing the memory change dynamically
Pages: Virtual Memory Blocks

- **Page faults**: if data not in memory, retrieve them from disk
  - huge miss penalty, thus pages should be fairly large (typical 4 - 16 KB; new desktop & servers support 32 & 64 KB; new embedded systems are toward 1 KB)
  - reducing page faults is important (LRU is worth, allow full associative placement)
  - can handle the faults in software instead of hardware
  - using write-through is too expensive (writes to disk take millions of processor cycles) so we use **write-back (copy back)**

- **Address translation (memory mapping)**: CPU produces virtual address, translated by hardware / software to **physical address**, which can be used to access main memory
How VM Differs from Memory Caches

- Much higher miss penalty (millions of cycles)!
  - Large pages [equivalent of cache line] (4 KB to MBs)
  - Associative mapping of pages (typically fully associative)
  - Software handling of misses (but HW handles hits!)
  - Write-through never used, only write-back

- Many pages
  - With 64 Mbyte memory, 4 Kbytes/pages, have 16 pages
    - It’s not practical to have 16K comparitors
    - Nor to have software do many comparisons
  - How can we get virtual memory with full associativity?
Implementation: Page Tables

- **Page table**: resides in main memory, pointed to by **page table register**
  - Indexed with page number from virtual address and contains corresponding physical page number, and valid bit to indicate if page is in main memory
  - Each program has its own page table
Here:
Virtual address space: $2^{32}$ bytes (4 GB)
Physical address space: $2^{30}$ bytes (memory up to 1 GB)
Number of entries in page table: $2^{20}$
Each entry here needs only 19 bits, but typically rounds up to 32 bits, with extra bits used for additional information & protection
Page Tables: Page Faults, Table Sizes, & Write Scheme

- **Page fault**: if valid bit in page table is off, a page fault occurs, and operating system gets control to find page in disk and decides where to place page in main memory (LRU or approximate LRU if all pages in memory are in use)

- **Page table sizes**:
  - **Example**: 32-bit virtual address, 4 KB pages, 4 bytes per page table entry. Page table size = ?
  - **Solution**: Page offset = 12 bits; number of page table entries = $2^{32} / 2^{12} = 2^{20}$
  - Page table size = $2^{20} \times 4 = 4$ MB
  - Page table uses huge amount of memory --> techniques to limit page table size

- **Write-back**: used for virtual memory due to writes take millions of processor cycles; note also that disk transfer time is small compared with its access time (another advantage of write-back over write-through)

- A **dirty bit** is added to page table and set when page is first written, indicating the need to copy back when page is being replaced
TLB: Translation Lookaside Buffer

- Since page table memory: 1 memory access to get physical address + 1 access to get data --> slow
- Hence: locality of reference to page table --> A cache for address translations: TLB (Translation-Lookaside Buffer), which contains a subset of page mappings

TLB is a cache (subset) & therefore must have a tag field
On TLB miss, page table must then be examined

Typical TLB:
- 16 - 512 entries
- each block of 1 - 2 entries
- hit time of 0.5 - 1 cycle
- miss penalty of 10 - 100 cycles
- miss rate of 0.01 - 1%
How Do Cache and VM Fit Together

- Cache can be **virtually addressed**
  - The virtual address is broken into tag-index-offset to look up data in cache

- Or, **physically addressed**
  - The virtual address is first converted to a physical address (using the page table)
  - The physical address is used to find data in cache

- Virtually addressed caches are faster, but make sharing data between processes complicated.
- One compromise is to use virtual address for the index and physical address for the tag
Integrating Virtual Memory, TLBs, and Caches

TLB --> cache -->
Integrating Virtual Memory, TLBs, and Caches

Virtual address

TLB access

TLB hit?

Yes

Physical address

No

Write?

Yes

Write protection bit on?

No

Cache hit?

No

Cache miss stall while read block

Yes

Try to read data from cache

No

Cache hit?

Try to write data to cache

Yes

Cache miss stall while read block

No

Write data into cache, update the dirty bit, and put the data and the address into the write buffer

No

Cache miss stall while read block

Yes

Write protection bit on?

No

Cache hit?

Yes

Deliver data to the CPU

No

Cache hit?
Virtual memory allows sharing of main memory by several processes: need to protect reading / writing of data from one process to another (or to OS)

Hardware provide 3 basic capabilities:
- Support at least 2 modes: user process or kernel process (supervisor process, OS process, executive process)
- Provide a portion of CPU state that a user process can read but not write
- Provide mechanism for CPU to go from user to supervisor mode, & vice versa

Store page tables in OS’s address space, preventing user processes from changing them
Virtual Memory Key Points

- Virtual memory provides
  - protection
  - sharing
  - performance
  - illusion of large main memory

- Virtual Memory requires twice as many memory accesses, so we cache page table entries in the TLB

- Four things can go wrong on a memory access:
  - cache miss
  - TLB miss (but data is found in cache)
  - TLB miss and data misses in cache
  - page fault
Processor speeds continue to increase very fast
— much faster than either DRAM or disk access times
Design challenge: dealing with this growing disparity

Paged memory: miss rate < 0.0001% typically

Page faults take millions of processor cycles, OS will usually select another process to execute in the CPU during disk access

Virtual memory always use fully associative placement while set-associative placement often used by caches & TLBs

Segmentation (variable-size scheme) (vs fixed-size page scheme)

Cache & TLB: physically addressed (at least 2 cache accesses) vs virtually addressed (faster, but more complex)

A compromise is virtually addressed cache with physical tags
### Memory: Some Data

- **CPU vs DRAM**

#### 2006:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Typical for L1 cache</th>
<th>Typical for L2 cache</th>
<th>Typical for paged memory</th>
<th>Typical for TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>16 – 64 KB</td>
<td>2 – 3 MB</td>
<td>500 MB – 1 TB</td>
<td>0.25-16</td>
</tr>
<tr>
<td>Miss penalty (cycles)</td>
<td>10 - 25</td>
<td>100 - 1000</td>
<td>10 M – 100 M</td>
<td>10-1000</td>
</tr>
<tr>
<td>Miss rates</td>
<td>2 – 5%</td>
<td>0.1 – 2%</td>
<td>0.00001-0.0001%</td>
<td>0.01-2%</td>
</tr>
</tbody>
</table>
Chapter Summary

- Memory hierarchy: registers - cache - main memory - disk
- Memory hierarchy: exploiting locality to try achieve speed of highest level and size of lowest level
- SRAMs, DRAMs
- Cache: direct mapped, set associative, fully associative
- Write schemes: write-through, write-back
- Replacement strategy: LRU
- Memory interleaving to support caches
- Cache performance: model, effects of cache sizes, block sizes, associativity
- Multi-level caches
- Memory technology
- Virtual memory: page, page faults, write-back
- Page tables & address translation, TLBs
- Integrating virtual memory, TLBs, & caches