16.482 / 16.561 Computer Architecture and Design

Instructor: Dr. Michael Geiger

Spring 2014

Lecture 7:

Multiple issue and multithreading

Memory hierarchies

Lecture outline

- Announcements/reminders
  - HW 5 to be posted; due 4/10
- Today's lecture
  - Multiple issue and multithreading
  - Memory hierarchy design
  - Return midterm exams

Getting CPI below 1

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  - statically-scheduled superscalar processors,
  - dynamically-scheduled superscalar processors, and
  - VLIW (very long instruction word) processors
- 2 types of superscalar processors issue varying numbers of instructions per clock
  - use in-order execution if they are statically scheduled, or
  - out-of-order execution if they are dynamically scheduled

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications
  (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

New Approach: Mulithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks
- When switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Fine-Grained Multithreading

- Switch on each instruction
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage: Hide both short/long stalls
- Disadvantage: slows individual threads

Coarse-Grained Multithreading

- Switches only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn’t slow down individual thread
- Disadvantage: hard to overcome throughput losses on shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time

Do both ILP and TLP?

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP exploit TLP?
  - Functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

Simultaneous Multi-threading …

One thread, 8 units

Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨							🟨
2	🟨	🟨					🟨	
3				🟨	🟨			
4								
5								
6								
7	🟨			🟨		🟨		
8		🟨			🟨			
9				🟨				

Two threads, 8 units

Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨	🟦	🟦					🟨
2	🟨	🟨	🟦			🟦	🟨	
3	🟦			🟨	🟨			
4	🟦	🟦				🟦		
5		🟦						🟦
6								
7	🟨		🟦	🟨	🟦	🟨		
8		🟨		🟦	🟨	🟦		
9	🟦	🟦		🟨		🟦		

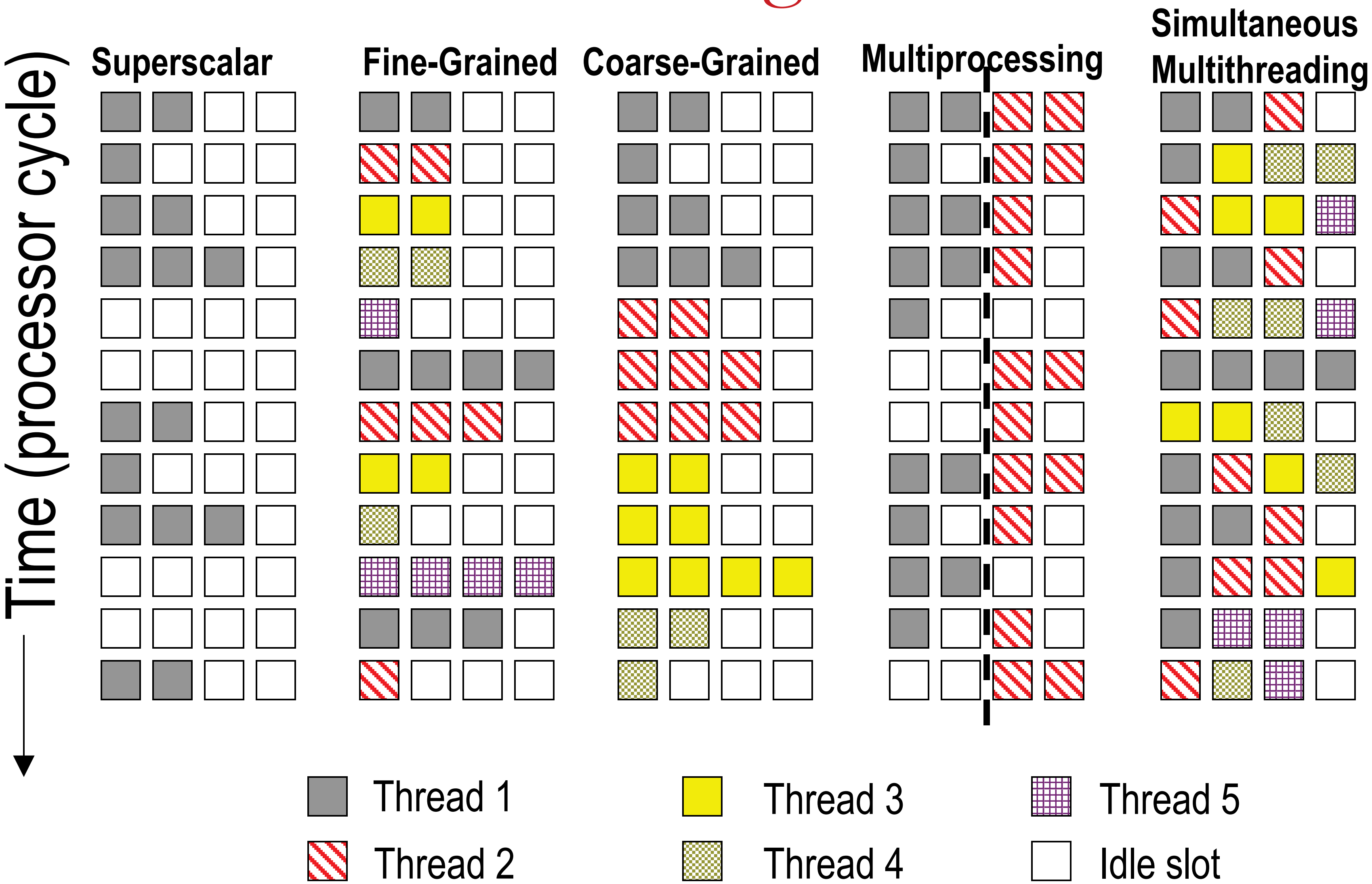
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Source: Microprocessor Report, December 6, 1999
"Compaq Chooses SMT for Alpha"

Multithreaded Categories



Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
    - A preferred thread approach sacrifices neither throughput nor single-thread performance (?)
    - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in
    - Instruction issue - more candidate instructions need to be considered
    - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

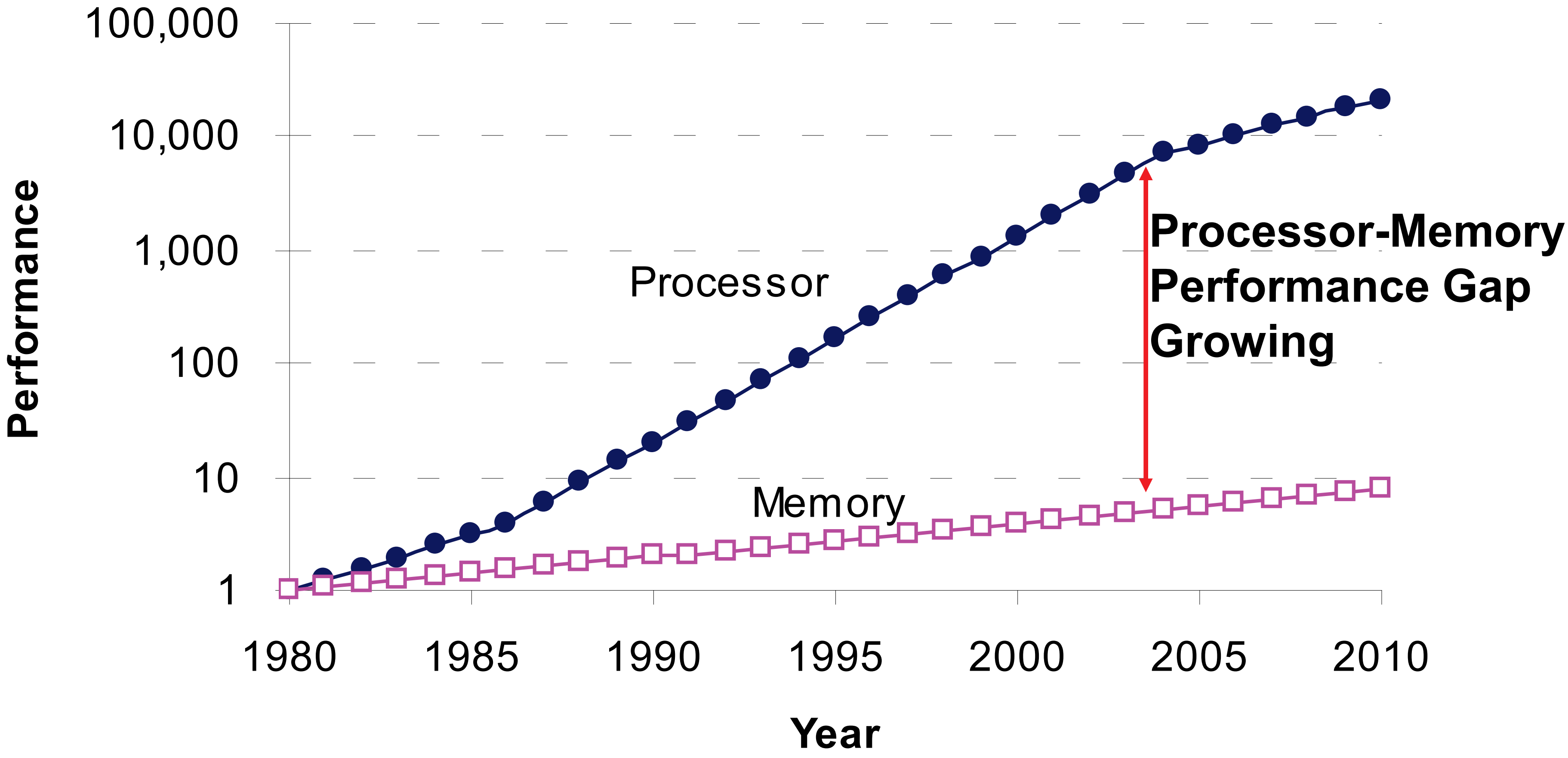
Multithreading examples

- Assume processor with following characteristics
  - 4 functional units
    - 2 ALU
    - 1 memory port (either load or store)
    - 1 branch
  - In-order scheduling
- Given 3 threads, show execution using
  - Fine-grained multithreading
  - Coarse-grained multithreading
    - Assume any stall longer than 2 cycles causes switch
  - Simultaneous multithreading
    - Thread 1 is preferred, followed by Thread 2 & Thread 3
- Assume any two instructions without stalls between them are independent

Motivating memory hierarchies

- Why do we need out-of-order scheduling, TLP, etc?
  - Data dependences cause stalls
  - The longer it takes to satisfy a dependence, the longer the stall and the harder we have to work to find independent instructions to execute
- Major source of stall cycles: memory accesses

Why care about memory hierarchies?



- Major source of stall cycles: memory accesses

Motivating memory hierarchies

- What characteristics would we like memory to have?
  - High capacity
  - Low latency
  - Low cost
- Can’t satisfy these requirements with one memory technology
- Solution: use a little bit of everything!
  - Small SRAM array(s) → **caches**
    - Small means fast and cheap
    - Typically at least 2 levels (often 3) of cache on chip
  - Larger DRAM array → **main memory** or **RAM**
    - Hope you rarely have to use it
  - Extremely large **disk**
    - Costs are decreasing at a faster rate than we fill them

Memory hierarchy analogy

- Looking for something to eat
  - First step: check the refrigerator
    - Find item → eat!
    - Latency = 1 minute
  - Second step: go to the store
    - Find item → purchase it, take it home, eat!
    - Latency = 20-30 minutes
  - Third step: grow food
    - Plant food, wait … wait some more … harvest, eat
    - Latency = ~250,000 minutes (~6 months)

Terminology

- Find data you want at a given level: **hit**
- Data is not present at that level: **miss**
  - In this case, check the next lower level
- **Hit rate**: Fraction of accesses that hit at a given level
  - (1 – hit rate) = **miss rate**
- Another performance measure: average memory access time

AMAT = (hit time) + (miss rate) x (miss penalty)

Average memory access time

- Given the following:
  - Cache: 1 cycle access time
  - Memory: 100 cycle access time
  - Disk: 10,000 cycle access time

What is the average memory access time if the cache hit rate is 90% and the memory hit rate is 80%?

AMAT example solution

- Key point: miss penalty = $\text{AMAT}_{\text{next level}}$
- Therefore:

$$
\begin{aligned}
\text{AMAT} &= (\text{HT}) + (\text{MR})(\text{MP}) \\
&= 1 + (0.1)(\text{AMAT}_{\text{memory}}) \\
&= 1 + (0.1)[100 + (0.2)(\text{AMAT}_{\text{disk}})] \\
&= 1 + (0.1)[100 + (0.2)(10{,}000)] \\
&= 1 + (0.1)[100 + 2000] \\
&= 1 + (0.1)[2100] \\
&= 1 + 210 = \mathbf{211\ cycles}
\end{aligned}
$$

Memory hierarchy operation

- We'd like most accesses to use the cache
  - Fastest level of the hierarchy
- But, the cache is much smaller than the address space
- Most caches have a hit rate > 80%
  - How is that possible?
  - Cache holds data most likely to be accessed

Principle of locality

- Programs don’t access data randomly—they display locality in two forms
  - **Temporal locality**: if you access a memory location (e.g., 1000), you are more likely to re-access that location than some random location
  - **Spatial locality**: if you access a memory location (e.g., 1000), you are more likely to access a location near it (e.g., 1001) than some random location

Basic cache design

- Each cache line consists of two main parts
  - Tag: Upper bits of memory address
    - Bits that uniquely identify a given block
    - Check tag on each access to determine hit or miss
  - Block: Actual data
    - Block sizes can vary—typically 32 to 128 bytes
    - Larger blocks exploit spatial locality
  - Status bits
    - Valid bit: Indicates if line holds valid data
    - We'll discuss dirty bit today; other bits in multiprocessors

Valid	Dirty	Tag	Block			
1	0	0x1000	0x00	0x12	0x34	0x56
0	0	0x2234	0xAB	0xCD	0x87	0xA9

Cache organization

- Cache consists of multiple tag/block pairs, called **cache lines**
  - Can search lines in parallel (within reason)
  - Each line also has a **valid bit**
  - Write-back caches have a **dirty bit**
- Note that block sizes can vary
  - Most systems use between 32 and 128 bytes
  - Larger blocks exploit spatial locality
  - Larger block size → smaller tag size

4 Questions for Memory Hierarchy

- 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)

Q1: Block placement (cont.)

- Three alternatives
  - Place block anywhere in cache: fully associative
    - Impractical for anything but very small caches
  - Place block in a single location: direct mapped
    - Line # = (Block #) mod (# lines) = index field of address
    - Block # = (Address) / (block size)
      - In other words, all bits of address except offset
  - Place block in a restricted set of locations: set associative
    - *n*-way set associative cache has *n* lines per set
    - Set # = (Block #) mod (# sets) = index field of address

Block placement example

- Given:
  - A cache with 16 lines numbered 0-15
  - Main memory that holds 2048 blocks of the same size as each cache block
- Determine which cache line(s) will be used for each of the memory blocks below, if the cache is (i) direct-mapped, or (ii) 4-way set associative
  - Block 0
  - Block 13
  - Block 249

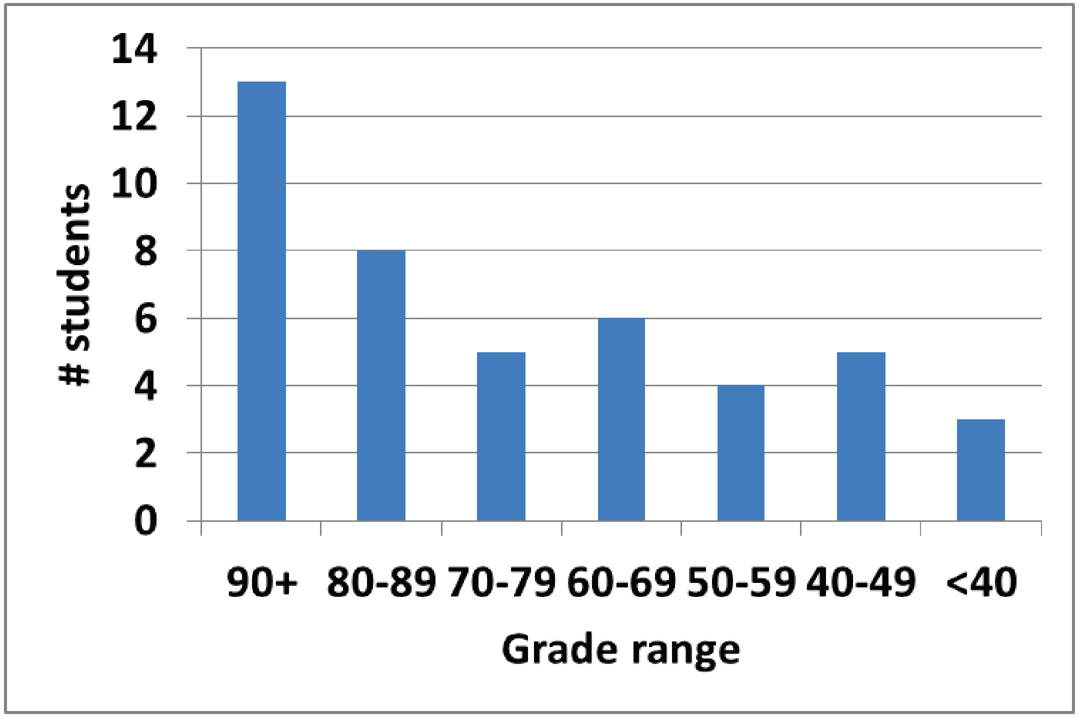
Solution

- Cache line # = (block #) mod (# sets)
  - "mod" = remainder of division (e.g., 5 mod 3 = 2)
  - Direct-mapped cache: 1 line per "set"
    - Block 0 → line (0 mod 16) = 0
    - Block 13 → line (13 mod 16) = 12
    - Block 249 → line (249 mod 16) = 9
      - 249 / 16 = 15 R9
  - 4-way SA cache: 4 lines per set → 4 total sets
    - Block 0 → set (0 mod 4) = lines 0-3
    - Block 12 → set (13 mod 4) = set 1 = lines 4-7
    - Block 249 → set (249 mod 4) = set 1 = lines 4-7

Exam stats & grade distribution

- Average: 83.7
- Median: 87
- Std. deviation: 16.1
- Max: 100 (x3)

- Question-by-question averages:
  - Q1: 10.7 / 12 (89%)
  - Q2: 13.2 / 14 (94%)
  - Q3: 15.9 / 18 (88%)
  - Q4: 12.3 / 14 (88%)
  - Q5: 17.3 / 20 (87%)
  - Q6: 14.3 / 22 (65%)



Final notes

- Next time
  - Finish memory hierarchy discussion
- Reminders
  - HW 5 to be posted; due 4/10