Quiz 6

Question 1: Multiple Choice

Average Score 2.6129 points

Consider a single 2-bit predictor that starts in the "weakly taken" state. After the following sequence of branches, which state will it be in (T == Taken, N == Not-taken):
T,T,T,NT,NT,T

Correct		Percent Answered
	Strongly not taken	1.613%
	weakly not taken	9.677%
☑	weakly taken	87.097%
	strongly taken.	1.613%
	Unanswered	0%

Quiz 6

Question 2: True/False

Average Score 1.90323 points

Squashing instructions and stalling the pipeline both result in increased CPI.

Correct	Answers	Percent Answered
☑	True	95.161%
	False	4.839%
	Unanswered	0%

Quiz 6

Question 3: True/False

Average Score 1.77419 point

Squashing can be used to resolve control hazards.

Correct	Answers	Percent Answered
☑	True	88.71%
	False	9.677%
	Unanswered	1.613%

Quiz 6

Question 4: True/False

Average Score 1.77419 points

Stalling cannot be used to resolve control hazards.

Correct	Answers	Percent Answered
	True	11.29%
☑	False	88.71%
	Unanswered	0%

Quiz 6

Question 5: True/False

Average Score 1.67742 points

Static branch predictors use tables of 2-bit counters.

Correct	Answers	Percent Answered
	True	16.129%
☑	False	83.871%
	Unanswered	0%

Quiz 6

Question 7: Calculated Numeric

Average Score 1.12903 points

If we double the number of pipeline stages in the MIPS 5-stage design by dividing each existing stage in half, what would the new branch delay penalty be (assume branches resolve at the end of decode)?

Correct	Answers	Percent Answered
☑	3	53.226%
	2	24.194%
	1	1.613%
	10	3.226%
	3 cycles	3.226%
	6	1.613%
	4	12.903%
	Unanswered	0%

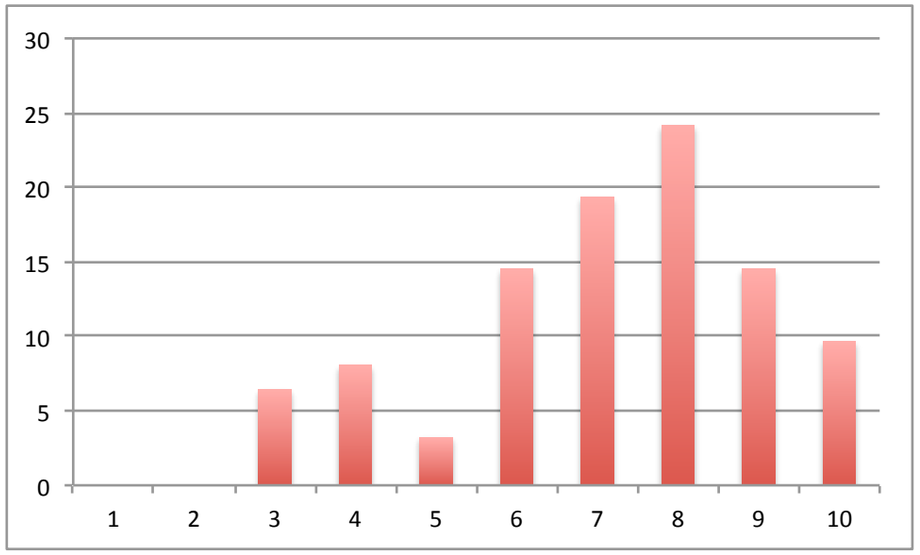
Quiz 6

Question 6: Calculated Numeric

Average Score 0 points

On a scale of 1-10 (1 being completely unfair and 10 being completely fair), how fair was the midterm?

Correct	Answers	Percent Answered
✓	3	6.452%
✓	10	9.677%
✓	8.333333333333333333333	1.613%
✓	7	19.355%
✓	6	14.516%
✓	5	3.226%
✓	4	8.065%
✓	9	14.516%
✓	8	22.581%
	Unanswered	0%



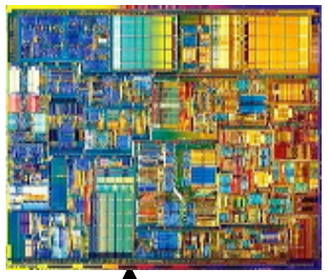
The Memory Hierarchy

In the book: 5.1-5.3, 5.7, 5.10

Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance (and other metrics) varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer systems?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a programs performance by considering the CPU running it?
  - How do other system components impact program performance?

Memory



CPU

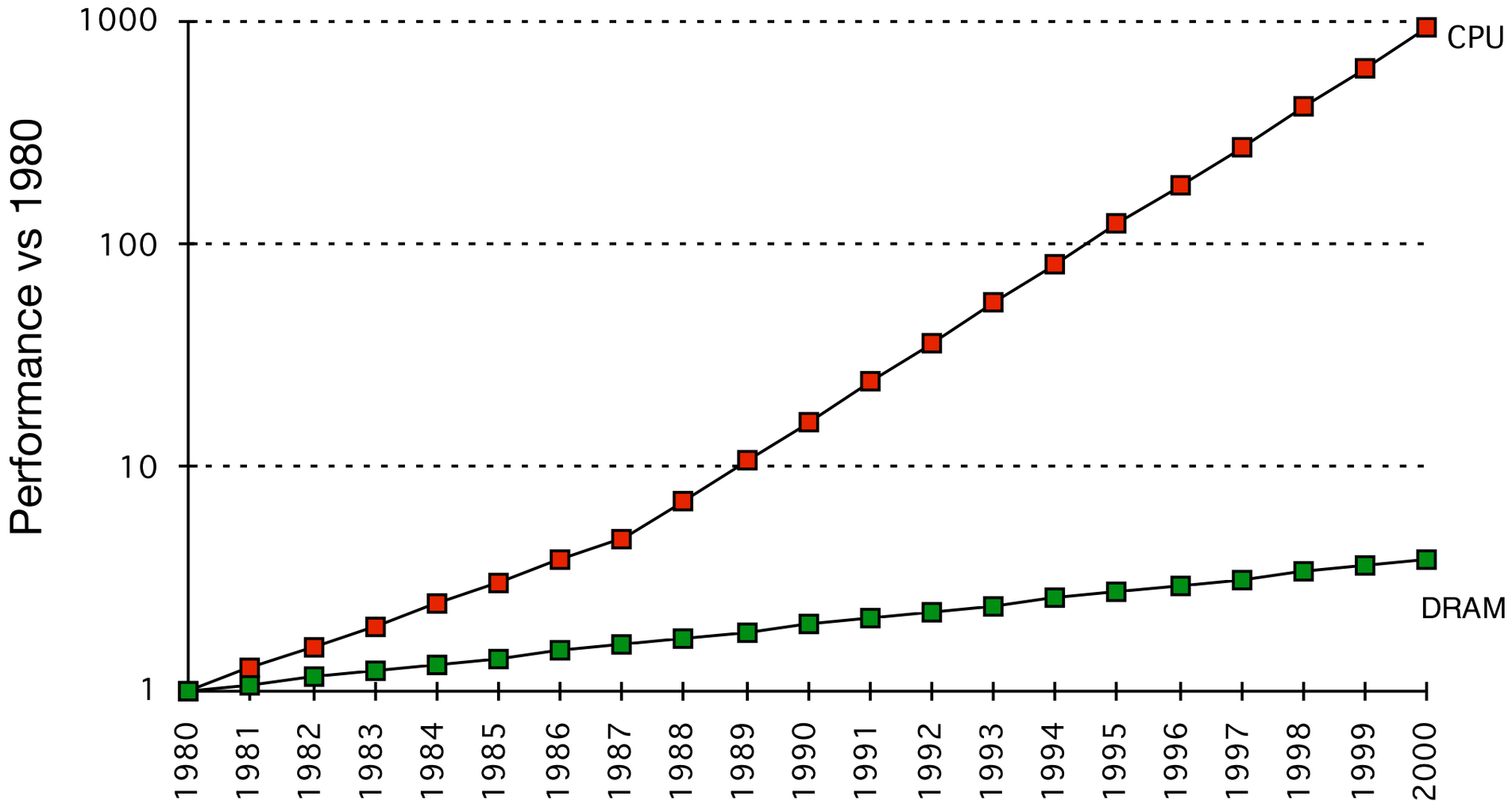
Abstraction: Big array of bytes

Main points for today

- What is a memory hierarchy?
- What is the CPU-DRAM gap?
- What is locality? What kinds are there?
- Learn a bunch of caching vocabulary.

Processor vs Memory Performance

- Memory is very slow compared to processors.



Memory's impact

M = % mem ops

Mlat (cycles) = average memory latency

BCPI = base CPI with single-cycle data memory

CPI =

Memory's impact

M = % mem ops
Mlat (cycles) = average memory latency
TotalCPI = BaseCPI + M*Mlat

Example:

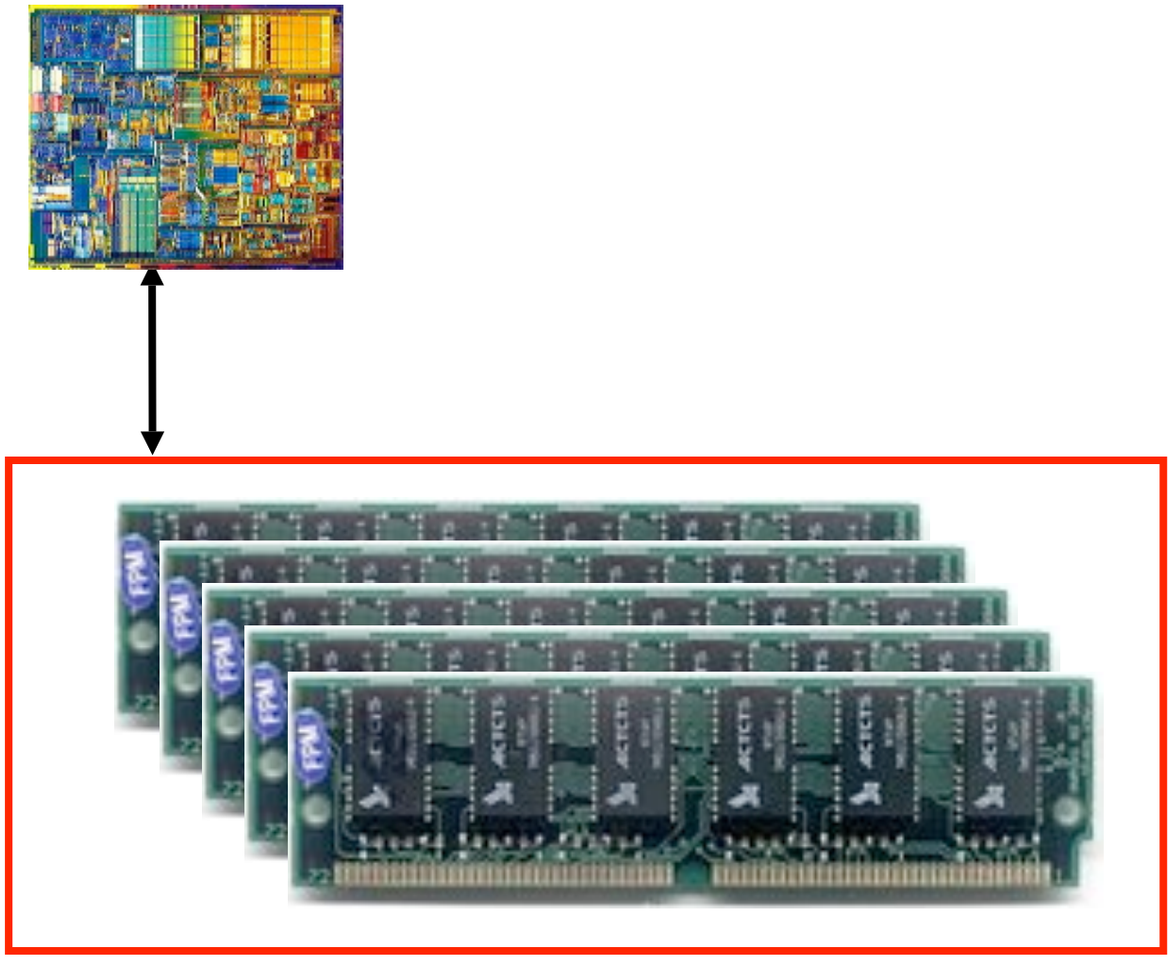
BaseCPI = 1; M = 0.2; Mlat = 240 cycles

TotalCPI = 49

Speedup = 1/49 = 0.02 => 98% drop in performance

Remember!: Amdahl's law does not bound the slowdown.
Poor memory performance can make your program arbitrarily slow.

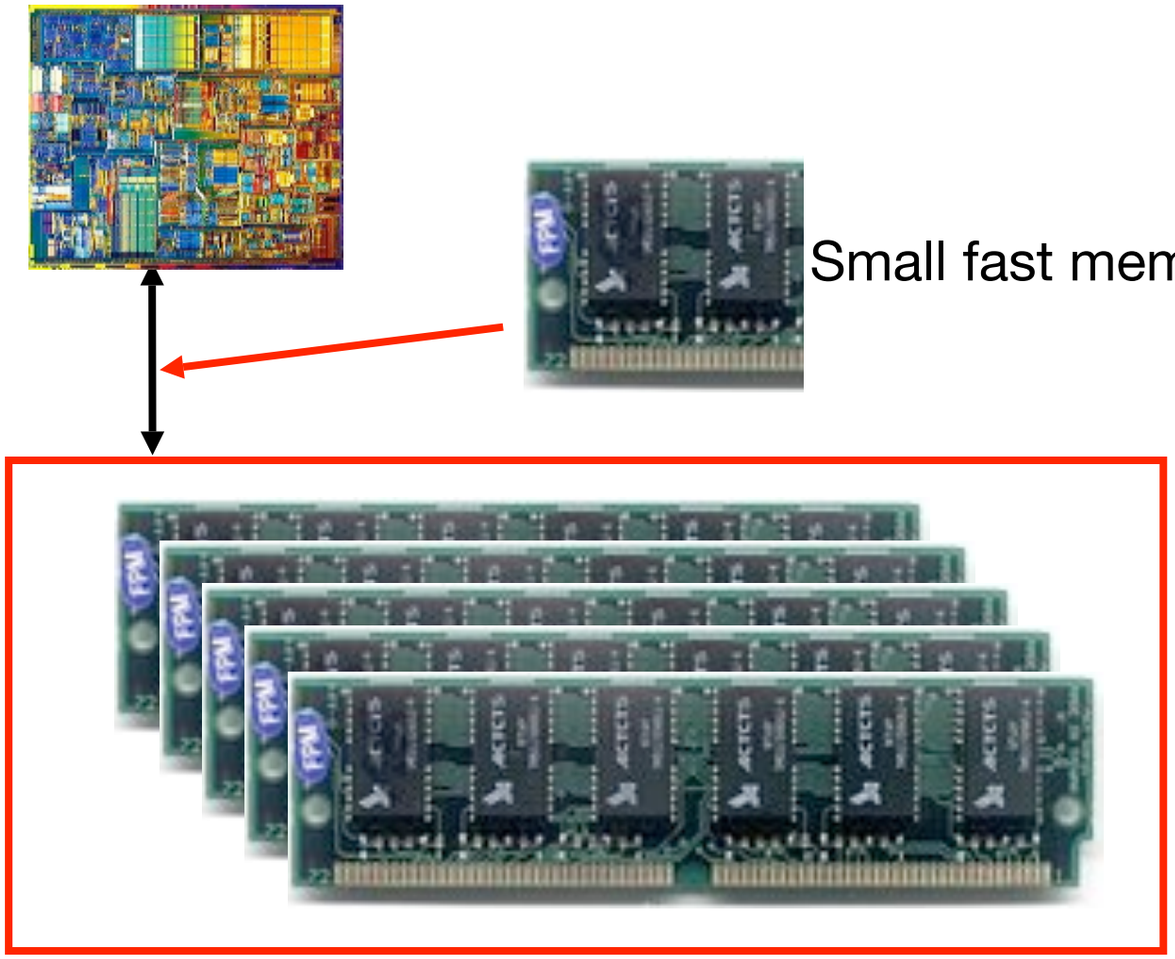
Memory Cache



Big slow memory

- Memory cost
  - >> capacity -> more $$
  - >> speed/bw -> more $$
  - >> speed -> larger (less dense)
- Build several memories with different trade-offs
- How do you use it? Build a “memory hierarchy”
- What should it mean for the memory abstraction?

Memory Cache

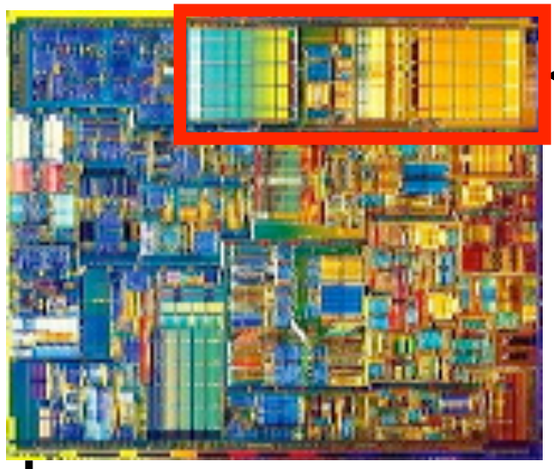


Small fast memory (a “cache”)

Big slow memory

- Memory cost
  - >> capacity -> more $$
  - >> speed/bw -> more $$
  - >> speed -> larger (less dense)
- Build several memories with different trade-offs
- How do you use it? Build a “memory hierarchy”
- What should it mean for the memory abstraction?

A typical memory hierarchy



on-chip cache
KBs

Cost

Access time

< 1ns

off-chip cache
MBs

2.5 $/MB

5ns

main memory
GBs

0.07 $/MB

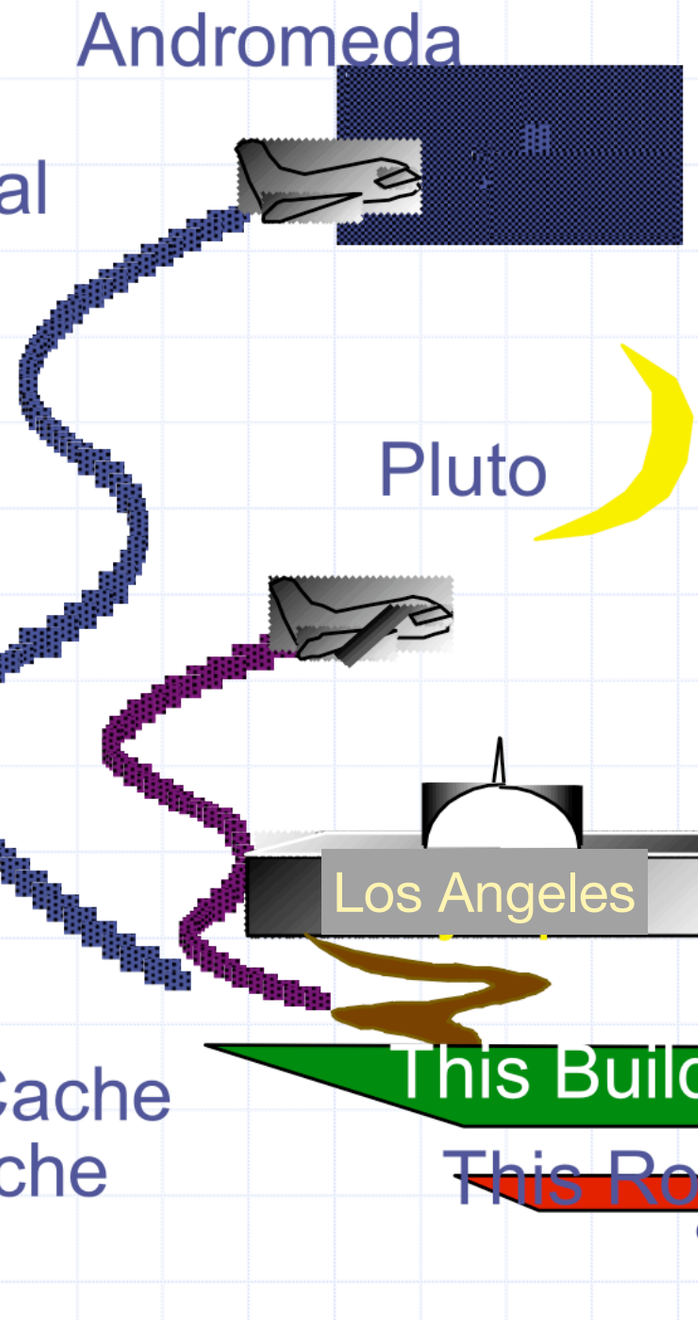
60ns

Disk
TBs

0.0004 $/MB

10,000,000ns

How far away is the data?



10^9	Tape /Optical Robot		2,000 Years
10^6	Disk		2 Years
100	Memory		1.5 hr
10	On Board Cache		10 min
2	On Chip Cache		
1	Registers		1 min

Why should we expect caching to work?

- Why did branch prediction work?

Why should we expect caching to work?

- Why did branch prediction work?
- Where is memory access predictable
  - Predictably accessing the same data
    - In loops: for(i = 0; i < 10; i++) {s += foo[i];}
    - foo = bar[4 + configuration_parameter];
  - Predictably accessing different data
    - In linked lists: while(l != NULL) {l = l->next;}
    - In arrays: for(i = 0; i < 10000; i++) {s += data[i];}
    - structure access: foo(some_struct.a, some_struct.b);

The Principle of Locality

- "Locality" is the tendency of data access to be predictable. There are two kinds:
  - Spatial locality: The program is likely to access data that is close to data it has accessed recently
  - Temporal locality: The program is likely to access the same data repeatedly.

Locality in Action

- Label each access with whether it has temporal or spatial locality or neither
  - 1
  - 2
  - 3
  - 10
  - 4
  - 1800
  - 11
  - 30
  - 1
  - 2
  - 3
  - 4
  - 10
  - 190
  - 11
  - 30
  - 12
  - 13
  - 182
  - 1004

Locality in Action

- Label each access with whether it has temporal or spatial locality or neither
  - 1 n
  - 2 s
  - 3 s
  - 10 n
  - 4 s
  - 1800 n
  - 11 s
  - 30 n
  - 1 t
  - 2 s, t
  - 3 s,t
  - 4 s,t
  - 10 s,t
  - 190 n
  - 11 s,t
  - 30 s
  - 12 s
  - 13 s
  - 182 n?
  - 1004 n

Locality in Action

- Label each access with whether it has temporal or spatial locality or neither
  - 1 n
  - 2 s
  - 3 s
  - 10 n
  - 4 s
  - 1800 n
  - 11 s
  - 30 n
  - 1 t
  - 2 s, t
  - 3 s,t
  - 4 s,t
  - 10 s,t
  - 190 n
  - 11 s,t
  - 30 s
  - 12 s
  - 13 s
  - 182 n?
  - 1004 n

There is no hard and fast rule here. In practice, locality exists for an access if the cache performs well.

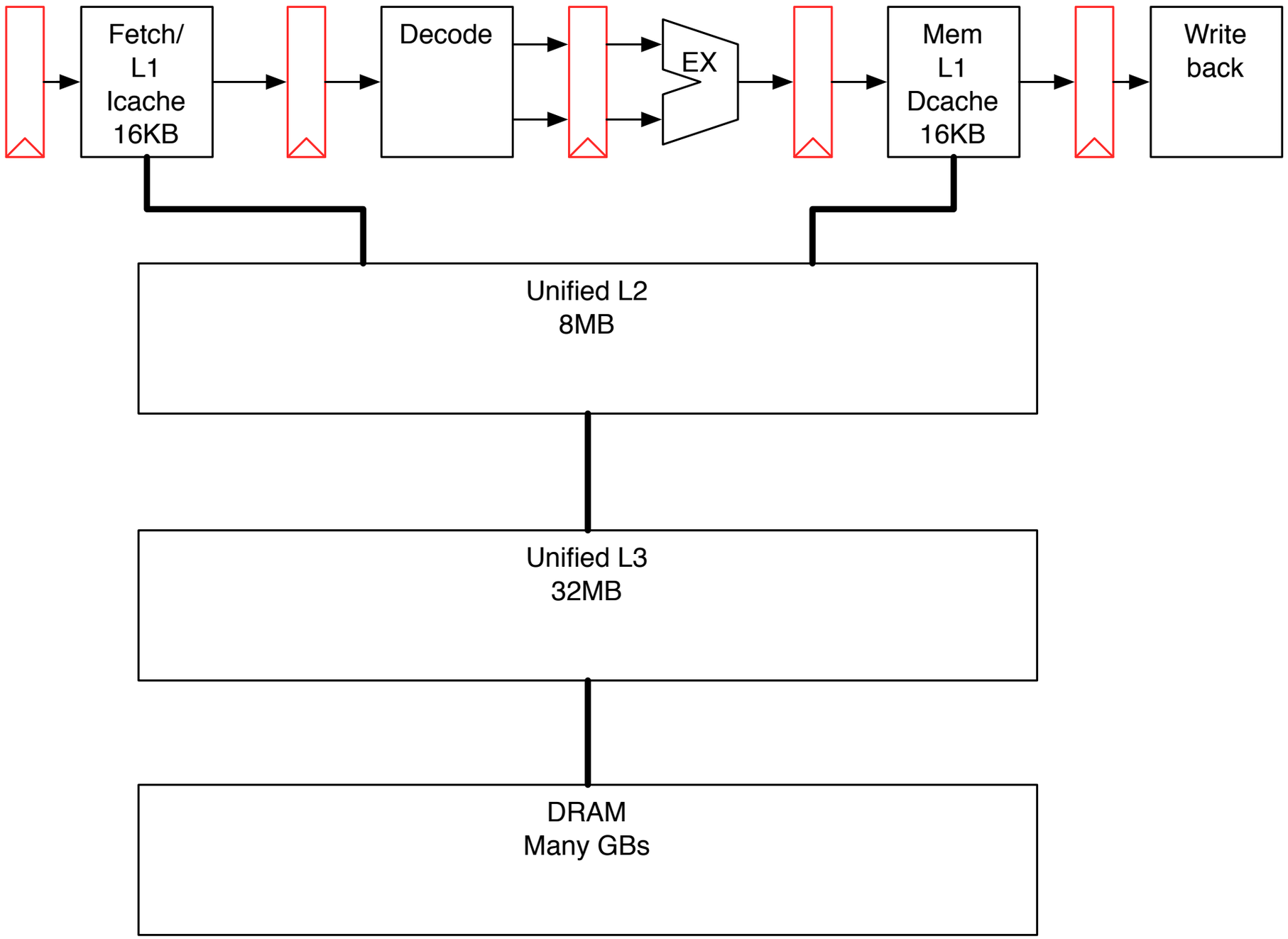
Cache Vocabulary

- *Hit* - The data was found in the cache
- *Miss* - The data was not found in the cache
- *Hit rate* - hits/total accesses
- *Miss rate* = 1- Hit rate
- *Locality* - see previous slides
- *Cache line* - the basic unit of data in a cache. generally several words.
- *Tag* - the high order address bits stored along with the data to identify the actual address of the cache line.
- *Hit time* -- time to service a hit
- *Miss time* -- time to service a miss (this is a function of the lower level caches.)

Cache Vocabulary

- There can be many caches stacked on top of each other
- if you miss in one you try in the "*lower level cache*" Lower level, mean higher number
- There can also be separate caches for data and instructions. Or the cache can be "*unified*"
- In the 5-stage MIPS pipeline
  - the L1 data cache (d-cache) is the one nearest processor. It corresponds to the "data memory" block in our pipeline diagrams
  - the L1 instruction cache (i-cache) corresponds to the "instruction memory" block in our pipeline diagrams.
  - The L2 sits underneath the L1s.
  - There is often an L3 in modern systems.

Typical Cache Hierarchy



Data vs Instruction Caches

- Why have different I and D caches?

Data vs Instruction Caches

- Why have different I and D caches?
  - Different areas of memory
  - Different access patterns
    - I-cache accesses have lots of spatial locality. Mostly sequential accesses.
    - I-cache accesses are also predictable to the extent that branches are predictable
    - D-cache accesses are typically less predictable
- Not just different, but often across purposes.
  - Sequential I-cache accesses may interfere with the data the D-cache has collected.
  - This is “interference” just as we saw with branch predictors
- At the L1 level it avoids a structural hazard in the pipeline
- Writes to the I cache by the program are rare enough that they can be slow (i.e., self modifying code)