CprE 381: Computer Organization and Assembly Level Programming

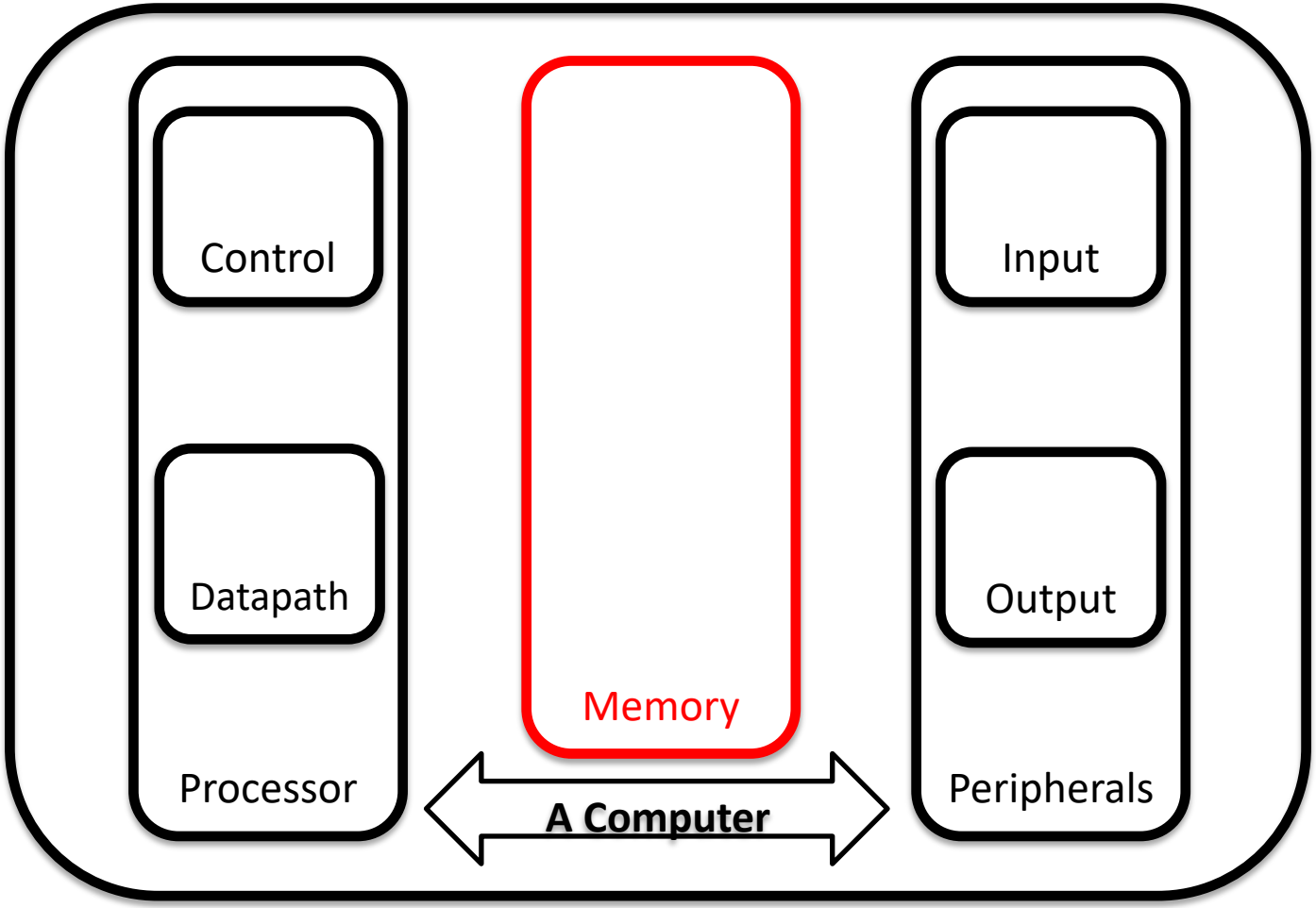
Cache Design

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- HW10 due on Mon April 16
  - Memory & Caches
- Part 3b due in lab next week

Remember the System View!



Review: Small or Slow

- Unfortunately there is a tradeoff between speed, cost and capacity

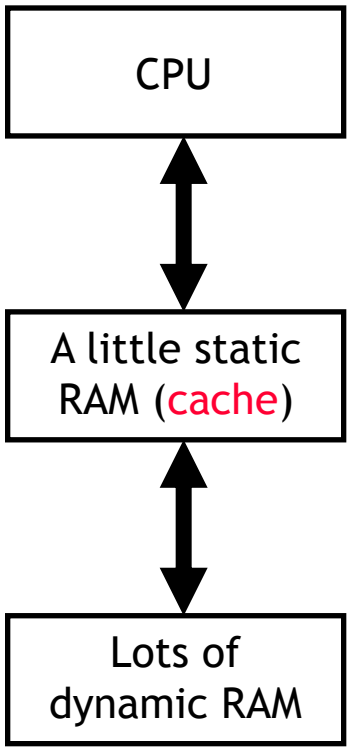
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- Fast memory is too expensive for most people to buy a lot of
- But dynamic memory has a much longer delay than other functional units in a datapath. If every `lw` or `sw` accessed dynamic memory, we'd have to either increase the cycle time or stall frequently
- Here are rough estimates of some current storage parameters

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$1	128KB-128MB
Dynamic RAM	100-200 cycles	~$0.005	256MB-32GB
Hard disks	10,000,000 cycles	~$0.00005	256GB-4TB

Review: Introducing Caches

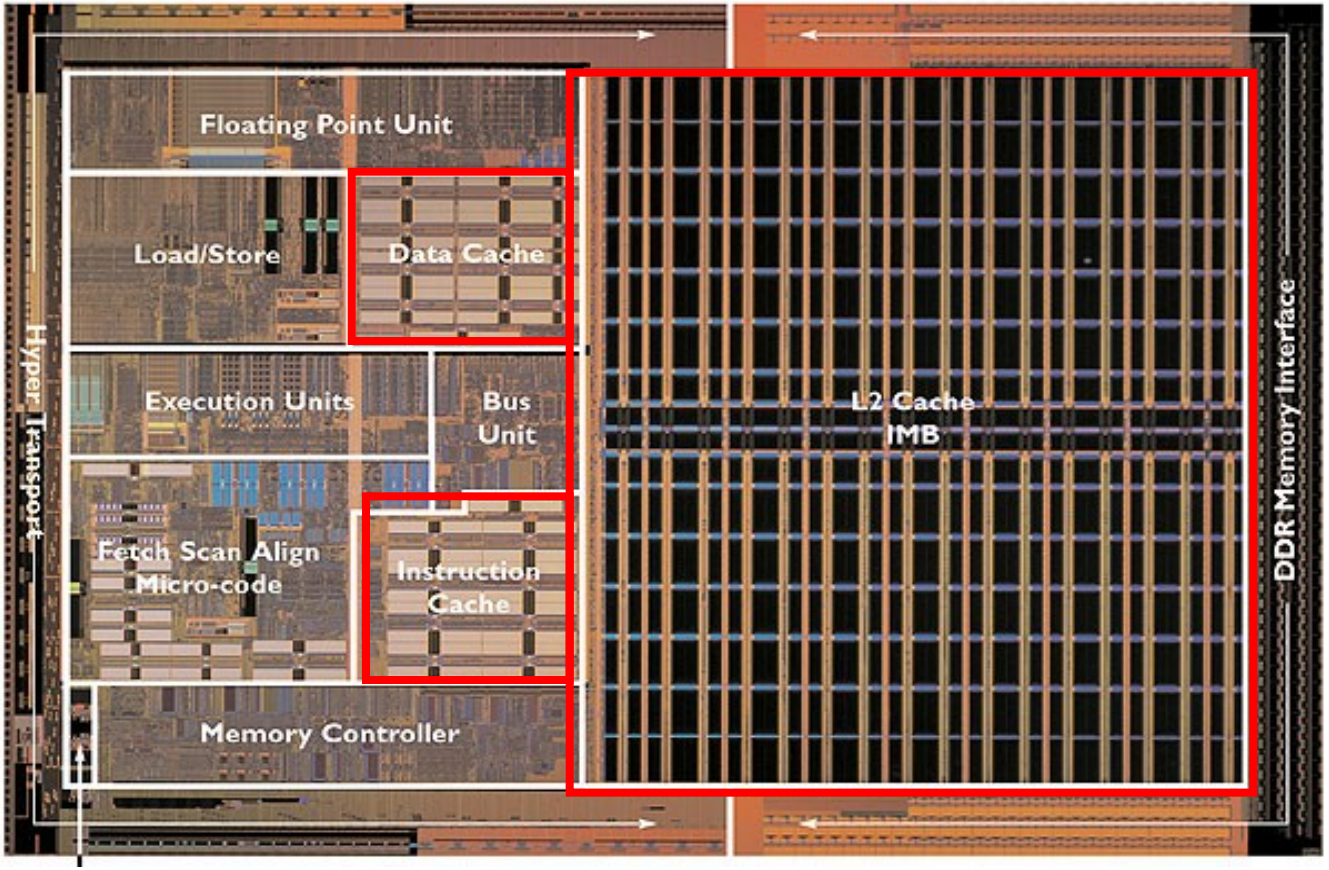
- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- We do this by introducing a cache, which is a small amount of fast, expensive memory
  - The cache goes between the processor and the slower, dynamic main memory
  - It keeps a copy of the most frequently used data from the main memory
- Memory access speed increases overall, because we've made the common case faster
  - Reads and writes to the most frequently used addresses will be serviced by the cache
  - We only need to access the slower main memory for less frequently used data



Review: The Principle of Locality

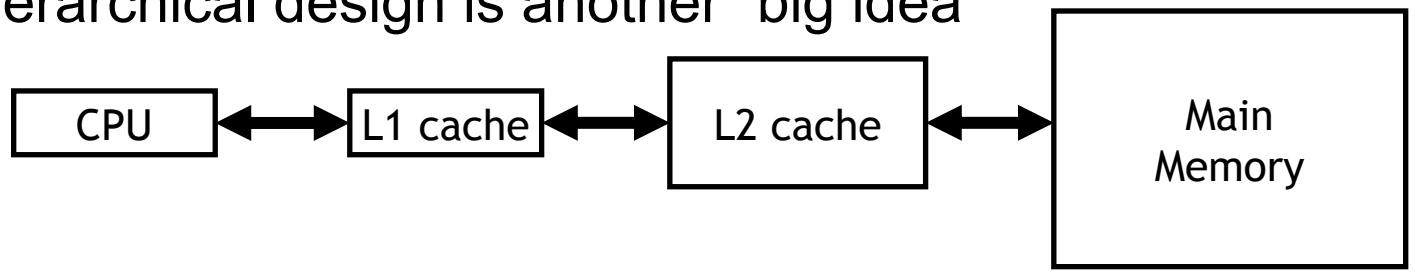
- It's usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

A Real Design (AMD Opteron)

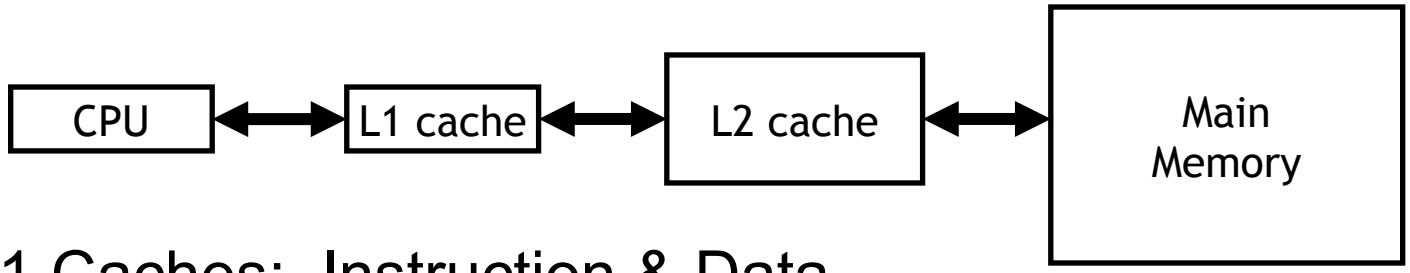


First Observations

- Split Instruction/Data caches:
  - Pro: No structural hazard between IF & MEM stages
    - A single-ported unified cache stalls fetch during load or store
  - Con: Static partitioning of cache between instructions & data
    - Bad if working sets dynamically unequal: *e.g.,* $_{\text{code}}$/**DATA** then **CODE**/$_{\text{data}}$
- Cache Hierarchies:
  - Trade-off between access time & hit rate
    - L1 cache can focus on fast access time (with okay hit rate)
    - L2 cache can focus on good hit rate (with okay access time)
  - Such hierarchical design is another "big idea"

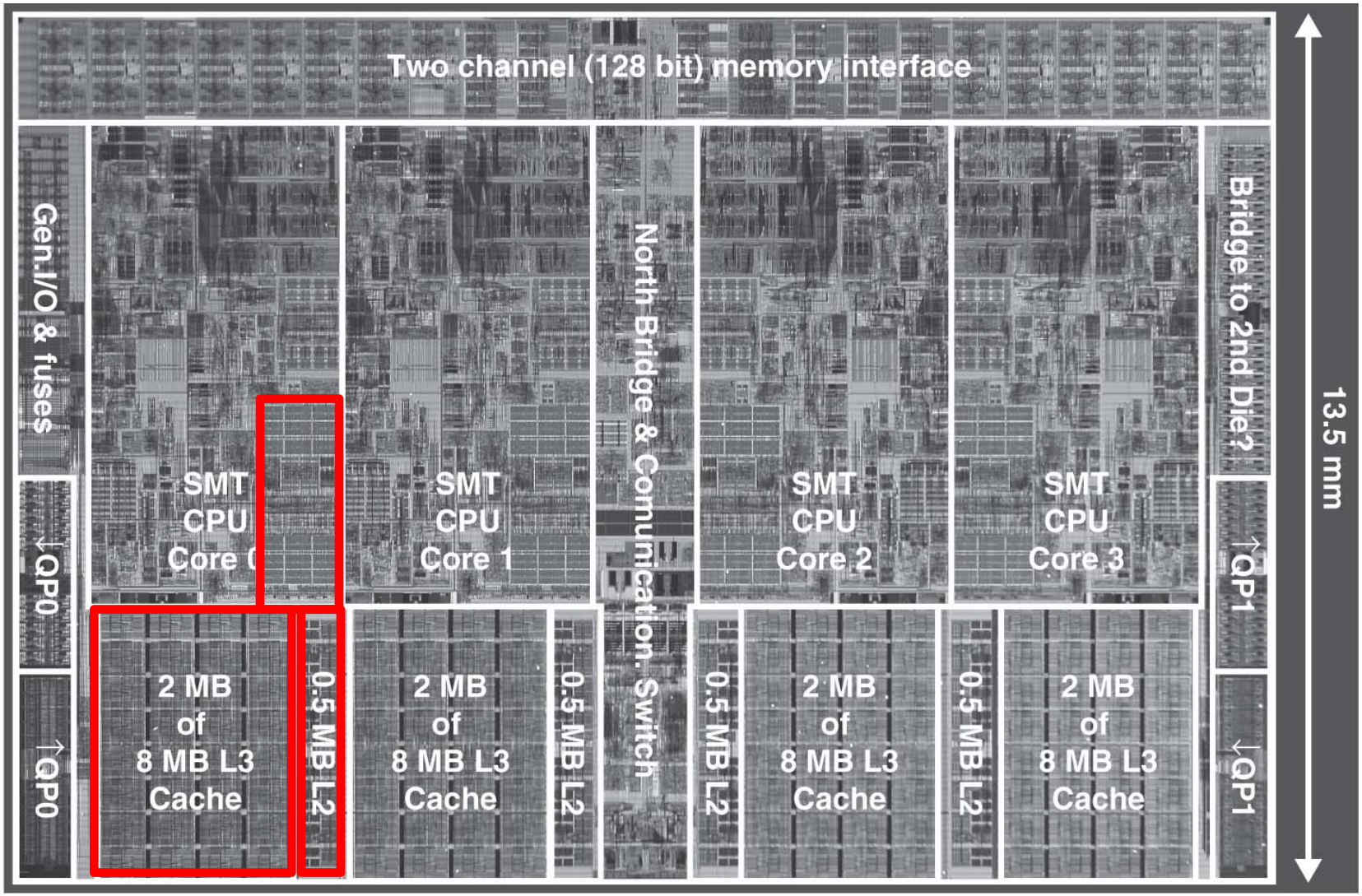


Opteron Vital Statistics

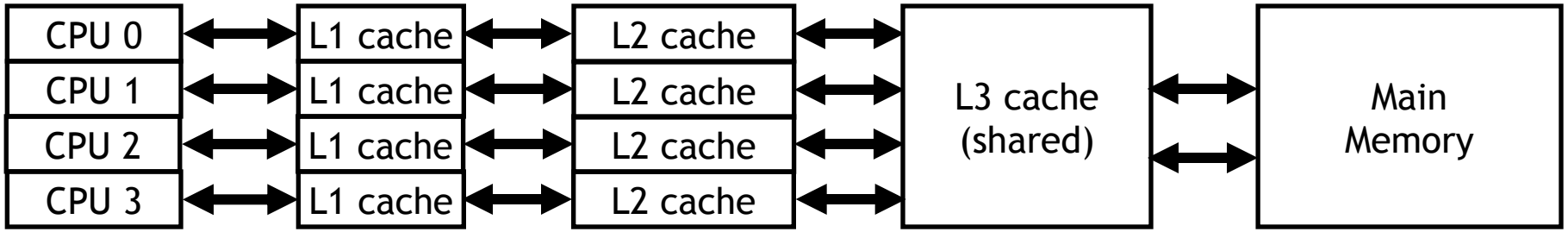


- L1 Caches: Instruction & Data
  - 64 kB
  - 64 byte blocks
  - 2-way set associative
  - 2 cycle access time
- L2 Cache:
  - 1 MB
  - 64 byte blocks
  - 4-way set associative
  - 16 cycle access time (total, not just miss penalty)
- Memory
  - 200+ cycle access time

Another Real Design (Intel Core i7)



Core i7 Vital Statistics



- Split L1 instruction and data caches
  - 32 KB (per core)
  - 4-way set associative (instruction), 8-way set associative (data)
  - 64 byte blocks
  - Write-back, write-allocate
- Unified L2 cache
  - 512 KB (per core)
  - 8-way set associative
  - 64 byte blocks
- Unified L3 cache
  - 8192 KB (shared)
  - 16-way set associative
  - 64 byte blocks

Memory and Overall Performance

- Average Memory Access Time

AMAT = Hit time + (Miss rate x Miss penalty)

- How do cache hits and misses affect overall system performance?
  - Assuming a hit time of one CPU clock cycle, program execution will continue normally on a cache hit. (Our earlier computations always assumed one clock cycle for an instruction fetch or data access.)
  - For cache misses, we'll assume the CPU must stall to wait for a load from main memory.

- The total number of stall cycles depends on the number of cache misses *and* the miss penalty.

Memory stall cycles = Memory accesses x miss rate x miss penalty

- To include stalls due to cache misses in CPU performance equations, we have to add them to the "base" number of execution cycles

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

Performance Example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles

Memory stall cycles = Memory accesses x Miss rate x Miss penalty
= 0.33 `I` x 0.03 x 20 cycles
= 0.2 `I` cycles

- If `I` instructions are executed, then the number of wasted cycles will be 0.2 x `I`

- This code is 1.2 times slower than a program with a “perfect” CPI of 1!

Memory Systems are a Bottleneck

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

- Processor performance traditionally outpaces memory performance, so the memory system is often the system bottleneck
- For example, with a base CPI of 1, the CPU time from the last page is:

CPU time = (`I` + 0.2 `I`) x Cycle time

- What if we could *double* the CPU performance so the CPI becomes 0.5, but memory performance remained the same?

In-class Assessment!

Access Code: It'sA

Note: sharing access code to those outside of classroom or using access code while outside of classroom is considered cheating

Memory Systems are a Bottleneck

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

- Processor performance traditionally outpaces memory performance, so the memory system is often the system bottleneck
- For example, with a base CPI of 1, the CPU time from the last page is:

CPU time = (`I` + 0.2 `I`) x Cycle time

- What if we could *double* the CPU performance so the CPI becomes 0.5, but memory performance remained the same?

CPU time = (0.5 `I` + 0.2 `I`) x Cycle time

- The overall CPU time improves by just 1.2/0.7 = 1.7 times!
- Refer back to Amdahl's Law from textbook chapter 1
  - Speeding up only part of a system has diminishing returns

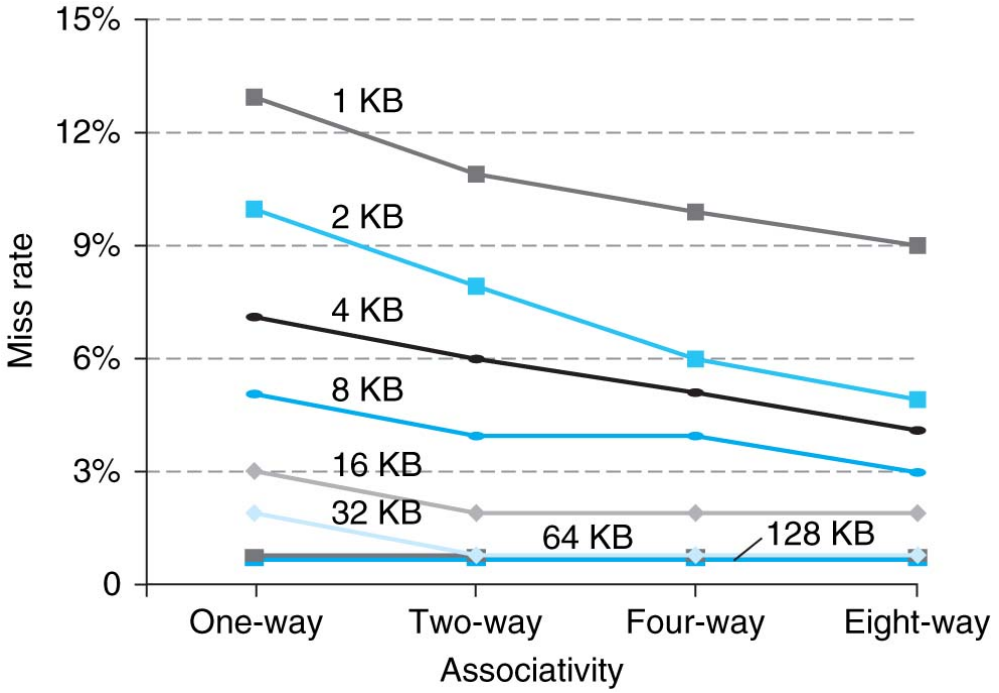
Comparing Cache Organizations

- Like many architectural features, caches are evaluated experimentally
  - As always, performance depends on the actual instruction mix, since different programs will have different memory access patterns
  - Simulating or executing real applications is the most accurate way to measure performance characteristics

- The graphs on the next few slides illustrate the simulated miss rates for several different cache designs
  - Again lower miss rates are generally better, but remember that the miss rate is just one component of average memory access time and execution time

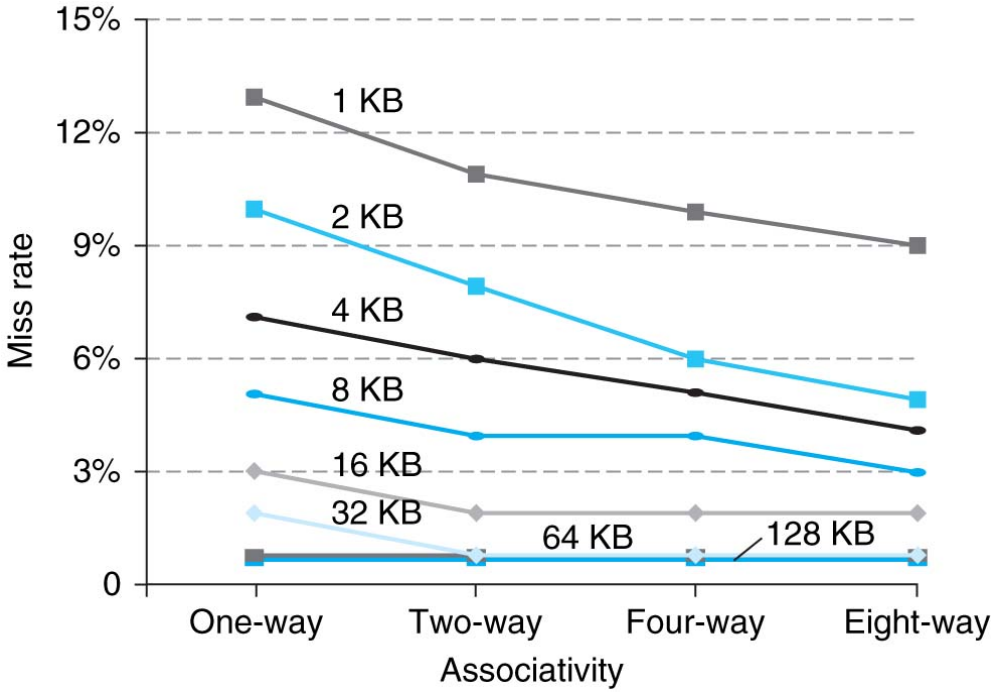
Associativity Tradeoffs and Miss Rates

- As we saw previously, higher associativity means more complex hardware
- But a highly-associative cache will also exhibit a lower miss rate
  - Each set has more blocks, so there's less chance of a conflict between two addresses which both belong in the same set
  - Overall, this will reduce AMAT and memory stall cycles
- Figure 5.36 of the textbook shows the miss rates decreasing as the associativity increases



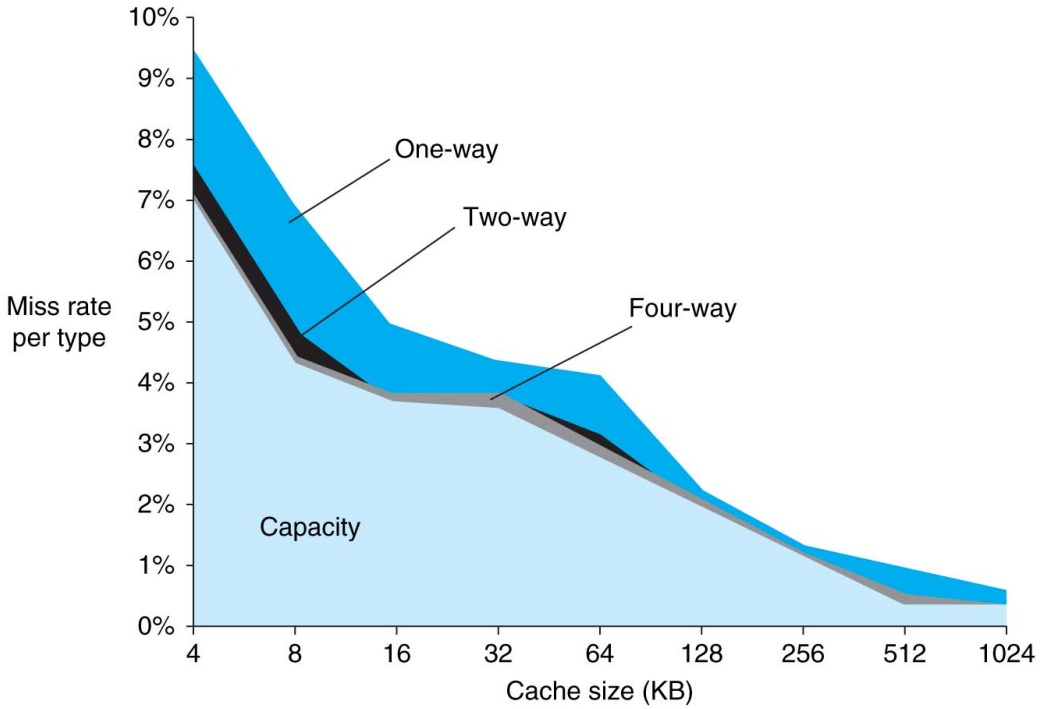
Cache Size and Miss Rates

- The cache size also has a significant impact on performance
  - The larger a cache is, the less chance there will be of a conflict
  - Again this means the miss rate decreases, so the AMAT and number of memory stall cycles also decrease
- Figure 5.36 also depicts the miss rate as a function of both the cache size and its associativity



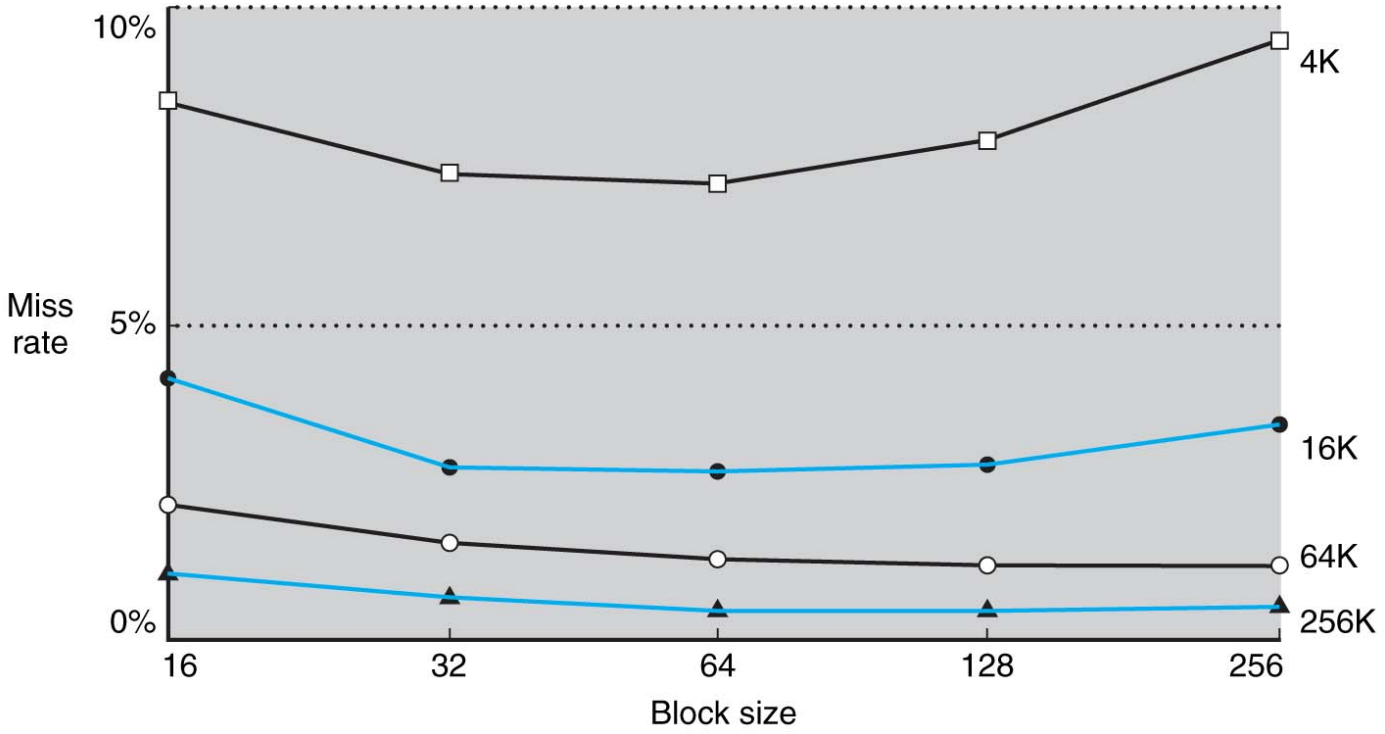
Categorizing Cache Misses

- **Compulsory** (initial misses when cache is empty)
- **Capacity** (cache is full, depends only on cache size)
- **Conflict** (two addresses corresponding to the same block, depends on cache size and associativity)



Block Size and Miss Rates

- Finally, Figure 5.11 shows miss rates relative to the block size and overall cache size
  - Smaller blocks do not take maximum advantage of spatial locality

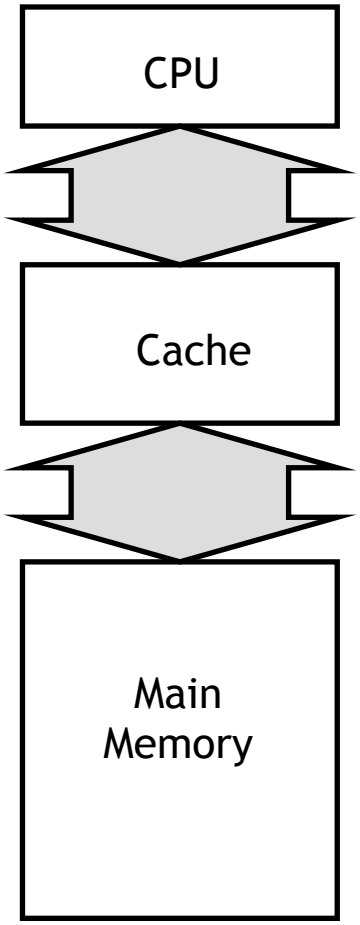


Basic Main Memory Design

- There are some ways the main memory can be organized to reduce miss penalties and help with caching
- For some concrete examples, let's assume the following three steps are taken when a cache needs to load data from the main memory
  1. It takes 1 cycle to send an address to the RAM
  2. There is a 15-cycle latency for each RAM access
  3. It takes 1 cycle to return data from the RAM
- In the setup shown here, the buses from the CPU to the cache and from the cache to RAM are all one word wide
- If the cache has one-word blocks, then filling a block from RAM (*i.e.*, the miss penalty) would take 17 cycles.

1 + 15 + 1 = 17 clock cycles

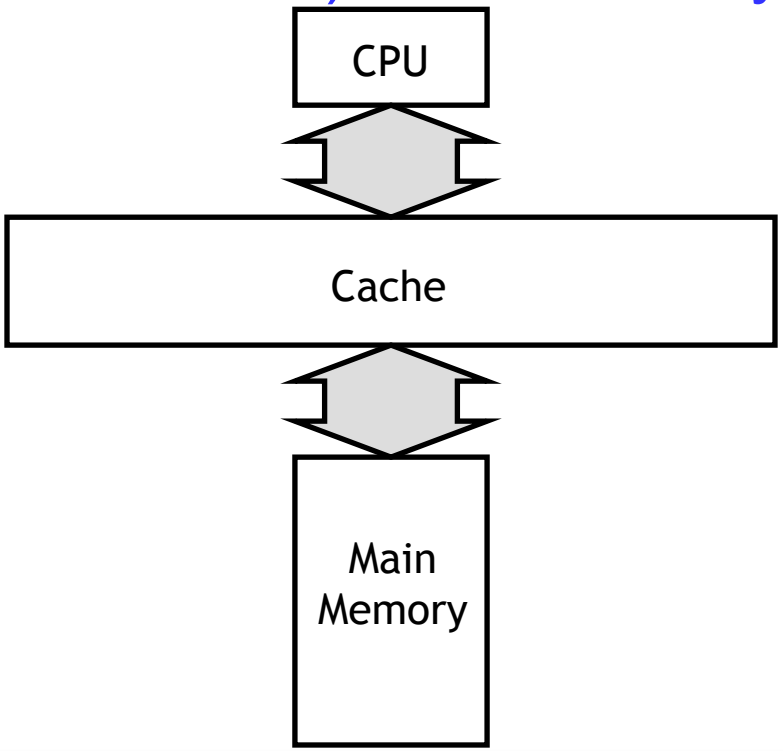
- The cache controller has to send the desired address to the RAM, wait and receive the data



Miss Penalties for Larger Cache Blocks

- If the cache has four-word blocks, then loading a single block would need four individual main memory accesses, and a miss penalty of 68 cycles!

4 x (1 + 15 + 1) = 68 clock cycles

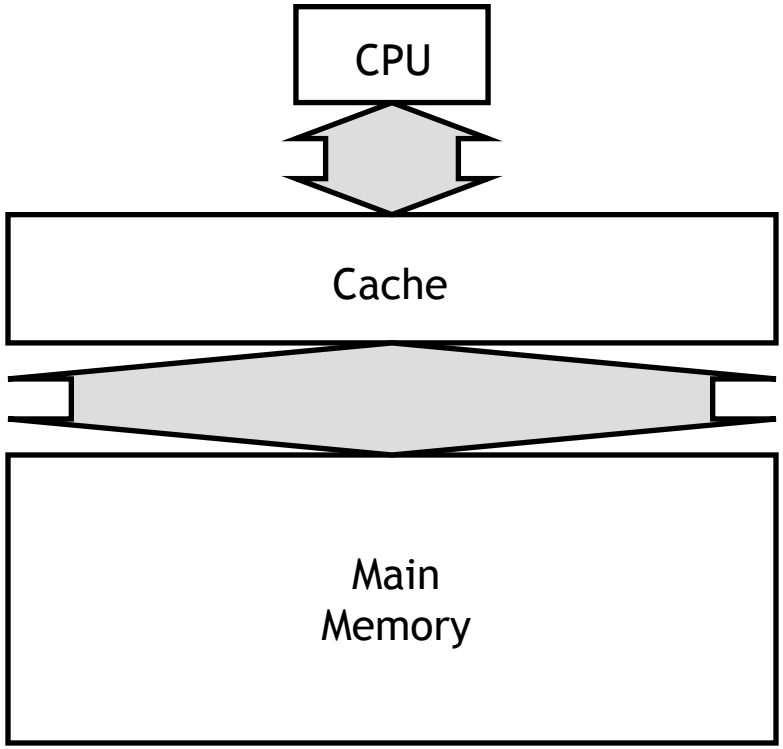


A Wider Memory

- A simple way to decrease the miss penalty is to widen the memory and its interface to the cache, so we can read multiple words from RAM in one shot
- If we could read four words from the memory at once, a four-word cache load would need just 17 cycles

  1 + 15 + 1 = 17 cycles

- The disadvantage is the cost of the wider buses – each additional bit of memory width requires another connection to the cache

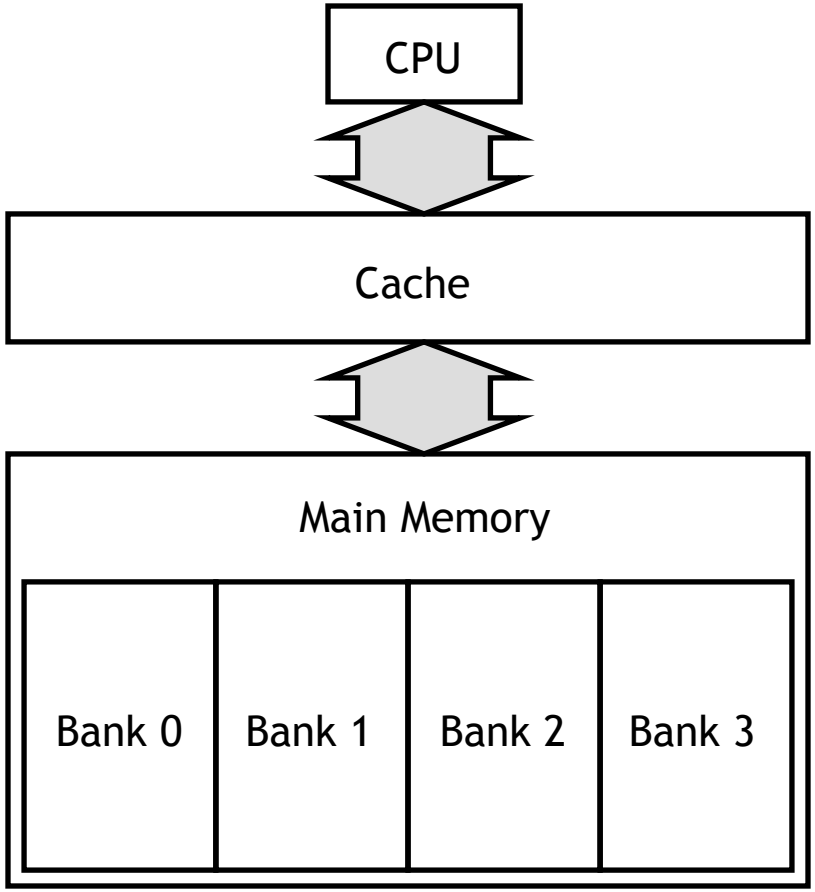


An Interleaved Memory

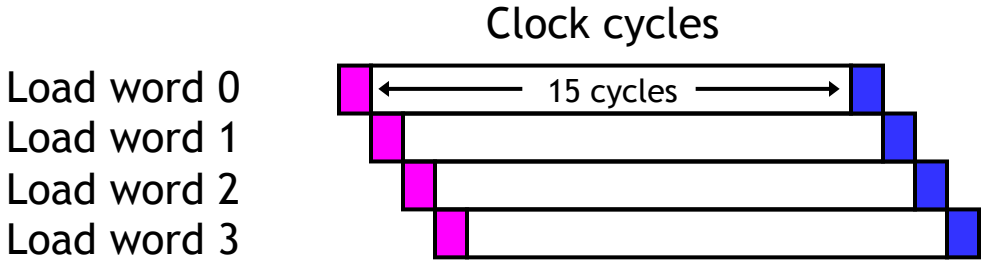
- Another approach is to interleave the memory, or split it into "banks" that can be accessed individually
- The main benefit is overlapping the latencies of accessing each word
- For example, if our main memory has four banks, each one byte wide, then we could load four bytes into a cache block in just 20 cycles

  1 + 15 + (4 x 1) = 20 cycles

- Our buses are still one byte wide here, so four cycles are needed to transfer data to the caches
- This is cheaper than implementing a four-byte bus, but not too much slower



Interleaved Memory Access



- Here is a diagram to show how the memory accesses can be interleaved
  - The magenta cycles represent sending an address to a memory bank
  - Each memory bank has a 15-cycle latency, and it takes another cycle (shown in blue) to return data from the memory
- This is the same basic idea as pipelining!
  - As soon as we request data from one memory bank, we can go ahead and request data from another bank as well
  - Each individual load takes 17 clock cycles, but four overlapped loads require just 20 cycles

Which is Better?

- Increasing block size can improve hit rate (due to spatial locality), but transfer time increases. Which cache configuration would be better?

	Cache #1	Cache #2
Block size	32-bytes	64-bytes
Miss rate	5%	4%

- Assume both caches have single cycle hit times. Memory accesses take 15 cycles, and the memory bus is 8-bytes wide:
  - i.e., an 16-byte memory access takes 18 cycles:

1 (send address) + 15 (memory access) + 2 (two 8-byte transfers)

recall: AMAT = Hit time + (Miss rate x Miss penalty)

Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - Akhilesh Tyagi (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)
  - Morgan Kaufmann