Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

$$$

$$$

Cache Memory 2

$$$

Schedule

- **This week**
  - Chapter 6 – Memory systems (caches)
- **Next Tuesday**
  - **Exam 2** – Tuesday, Nov 1st
- **Next Thursday**
  - Chapter 6 – Virtual memory

Exam 2

- Similar format as last time
  - Closed notes, closed book, no calculator, etc…
  - **I will provide Table 4.7 (MARIE ISA)**
- **Chapter 4 – On the exam!**
  - MARIE architecture
    - Major components and operation
  - MARIE programs
    - **“Write a complete program that does XYZ”**
    - Subroutines, indirect instructions, etc..
  - “You be the assembler”

Exam 2

- **Chapter 5 – On the exam!**
  - Endianness
  - Infix and postfix notation
  - Memory addressing modes
  - Pipelines
  - Instruction sets
    - 0-address machines (i.e. stack machines)
    - 1-address machines (i.e. accumulator machines)
    - 2-address and 3-address machines (general purpose register machines)
  - 7 different instruction types (data movement, arithmetic, etc…)

Recap – 50 word Problem

- In computer architecture, hazards are opportunities for data corruption and incorrect calculations if a naïve pipeline design does not detect specific error conditions and accommodate them, potentially by introducing delays ("stalls") in the pipeline.
- **What is a**
  - **Data hazard?**
  - **Structural hazard?**
  - **Control hazard?**

Recap – 50 word Problem

- Data hazards represent obstacles preventing perfect parallel execution of instructions, such as when one instruction depends on a result produced by a previous instruction that has not yet finished (*a data hazard*), when multiple instructions rely on the same hardware element like a shared memory (*a structural hazard*), or when the next pipeline instruction cannot be immediately determined due to a yet-unresolved branch (*a control hazard*).
  - 66 words

Recap - Cache

- **Which is bigger – a cache or main memory?**
  - Main memory
- **Which is faster to access – the cache or main memory?**
  - Cache – It is **smaller** (which is faster to search) and **closer** to the processor (signals take less time to propagate to/ from the cache)
- **Why do we add a cache between the processor and main memory?**
  - Performance – hopefully frequently-accessed data will be in the faster cache versus slower main memory

Recap – Cache

- **Which is manually controlled – a cache or a register?**
  - Cache is automatically controlled by hardware
  - Registers (typically) are manually controlled by the assembly language program
- **Suppose a program wishes to read from a particular memory address. Which is searched first – the cache or main memory?**
  - Search the cache first – otherwise, there's no performance gain

Recap – Cache

- **Suppose there is a cache miss (data not found) during a 1 byte memory read operation. How much data is loaded into the cache?**
  - Trick question – we always load data into the cache **1 block at a time**. (Block size varies – 64 bytes on a Core i7 processor)

Recap – Direct Mapped Cache Search

1. Take the main memory address of desired data
   1. Split into **tag**, **block**, and **offset** fields (varies by cache and block size)

2. Go to the indicated block in the cache

3. Does the tag saved in the cache match the search tag?
   Is the block marked as valid?
   1. Yes on both – we have a **cache hit**! ☺
      1. Retrieve the data (go to the byte/word indicated by offset)
   2. Otherwise, we have a **cache miss**! ☹
      1. Need to go to main memory and get the data
      2. Load in the full block from main memory into the cache

Example 1 – Hit Ratio

- Computer design
  - Main memory: 256 bytes
  - Direct mapped cache
  - Cache size: 4 blocks
  - Block size: 4 bytes
- Computer program accesses (in order) memory locations
  - 0xAA, F5, 53, 1C, 8A, 8B, 8C, CD, E4, E5, E7, E9, CF, D0
- Questions
  - **What is the hit ratio?**
  - **What are the final cache contents?**

Example 1 – Hit Ratio

- Step 1 – Determine partitioning of address bits
- Tag field: 4 bits
  - All remaining bits (memory addresses are 8 bits total)
- Block field: 2 bits
  - To select between 4 blocks in cache
- Offset field: 2 bits
  - To select between 4 bytes in each block

Example 1 – Hit Ratio

- Step 2 – Partition all addresses according to fields
  - AA = 1010 10 10
  - F5 = 1111 01 01
  - 53 = 0101 00 11
  - 1C = 0001 11 00
  - 8A = 1000 10 10
  - 8B = 1000 10 11
  - 8C = 1000 11 00
  - CD = 1100 11 01
  - E4 = 1110 01 00
  - E5 = 1110 01 01
  - E7 = 1110 01 11
  - E9 = 1110 10 01
  - CF = 1100 11 11
  - D0 = 1101 00 00

Example 1 – Hit Ratio

- Step 3 – Determine what other bytes are in the same block that is being accessed (vary the offset field)
  - Example: AA → 1010 10 00 *(A8)* to 1010 10 11 *(AB)*

- For each address accessed, the full block range is:
  - AA = A8 - AB
  - F5 = F4 – F7
  - 53 = 50 - 53
  - 1C = 1C – 1F
  - 8A = 88 – 8B
  - 8B = 88 – 8B
  - 8C = 8C – 8F
  - CD = CC – CF
  - E4 = E4 – E7
  - E5 = E4 – E7
  - E7 = E4 – E7
  - E9 = E8 – EB
  - CF = CC – CF
  - D0 = D0 – D3

Example 1 – Hit Ratio

- Draw a table to help keep track of cache contents
  - The actual *data* is irrelevant for this problem, and is not shown…

Block 0	Tag: Valid? Memory range:
Block 1	Tag: Valid? Memory range:
Block 2	Tag: Valid? Memory range:
Block 3	Tag: Valid? Memory range:

Example 1 – Hit Ratio

- Step 4 – Fill in the cache by processing each memory address in sequence
  - Load AA
    - **Block 2**, tag A → Miss, valid bit not set → load A8 – AB
  - Load F5
    - **Block 1**, tag F → Miss, valid bit not set → Load F4 – F7
  - Load 53
    - **Block 0**, tag 5 → Miss, valid bit not set → Load 50 – 53
  - Load 1C
    - **Block 3**, tag 1 → Miss, valid bit not set → Load 1C – 1F
  - Load 8A
    - **Block 2**, tag 8 → Miss, tag doesn't match → Load 88 – 8B (replaces A8-AB)
  - Load 8B
    - **Block 2**, tag 8 → Hit**!**
  - Load 8C
    - **Block 3**, tag 8 → Miss, tag doesn't match → Load 8C – 8F (replaces 1C-1F)

Example 1 – Hit Ratio

- Step 4 continued…
  - Load CD
    - Block **3**, tag C → Miss, tag doesn't match → Load CC-CF
  - Load E4
    - Block **1**, tag E → Miss, tag doesn't match → Load E4 – E7
  - Load E5
    - Block **1**, tag E → **Hit**!
  - Load E7
    - Block **1**, tag E → **Hit**!
  - Load E9
    - Block **2**, tag E → Miss, tag doesn't match → Load E8 – EB
  - Load CF
    - Block **3**, tag C → **Hit**!
  - Load D0
    - Block **0**, tag D → Miss, tag doesn't match → Load D0 – D3

Example 1 – Hit Ratio

- Step 5 – Count the number of hits and total accesses
  - 4 hits in 14 accesses: hit ratio = 28.57%

Block 0	Tag: D Valid? Yes Memory range stored here: D0-D3
Block 1	Tag: E Valid? Yes Memory range stored here: E4-E7
Block 2	Tag: E Valid? Yes Memory range stored here: E8-EB
Block 3	Tag: C Valid? Yes Memory range stored here: CC-CF

Recap - Cache Thrashing

- Main disadvantage of direct mapped cache
  - Each main memory block can only go one place in the cache
  - Possible to have "thrashing" (where the cache continually evicts and replaces blocks)
- Other (more sophisticated) cache mapping schemes prevent this kind of thrashing

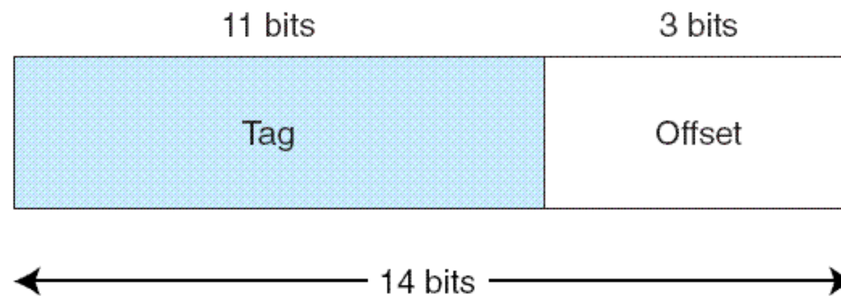
Fully Associative Cache

Fully Associative Cache

- Idea: instead of placing memory blocks in specific cache locations (based on memory address), allow a block to **go anywhere in the cache**
  - The cache would have to completely fill up before any blocks are evicted
- New design: **fully associative cache**
- Memory address is partitioned into only two fields
  - *Tag* and *Offset*

Fully Associative Cache

- Example for 14-bit memory addresses
  - Cache size: 16 blocks
  - Block size: 8 ($2^3 = 8$, thus 3 bits for offset)



- How to retrieve?
  - Search all tags in parallel!
  - This requires **special**, **costly hardware** (i.e. a **CAM**)

Fully Associative Cache

- The block that is evicted from a cache is the **victim block**
- Direct-Mapped cache
  - The victim is always the cache block with the matching block number
- Fully-Associated cache
  - No fixed mapping
  - How does hardware pick a victim?
- There are a number of ways to pick a victim
  - Discuss later in this chapter

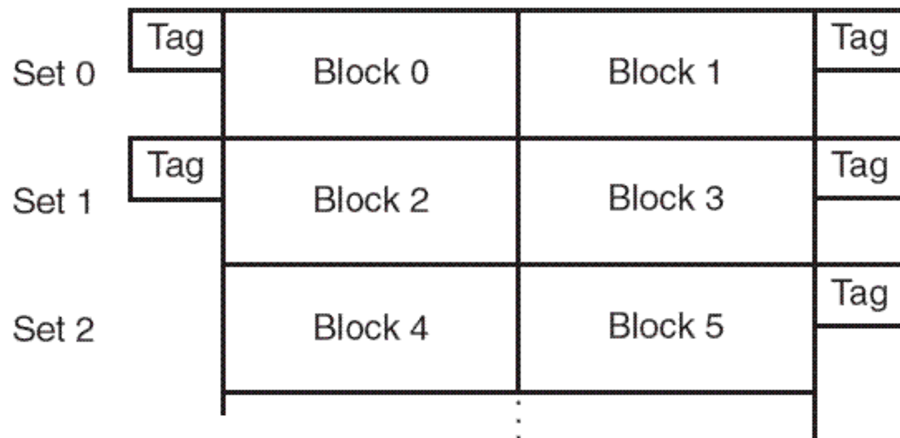
Set Associative Cache

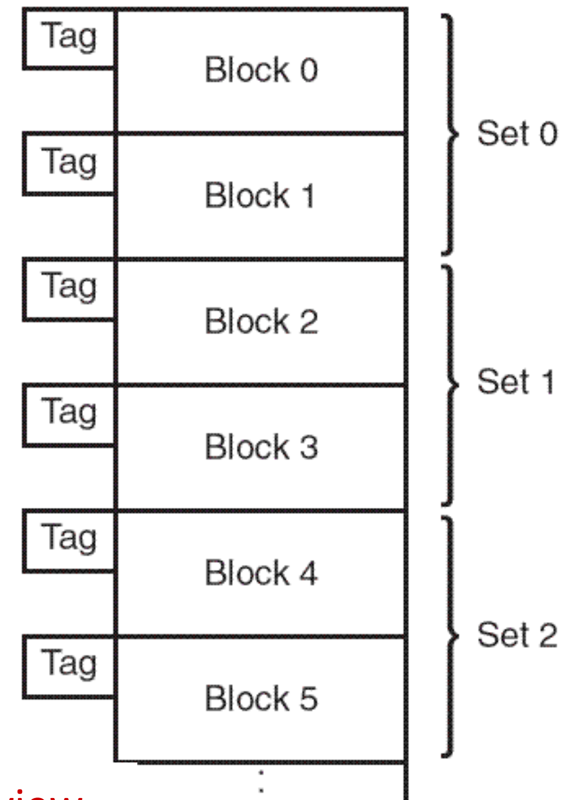
Set Associative Cache

- Set associative cache
  - Hybrid between direct mapped cache and fully associative cache
  - Reduces hardware complexity and improves performance
- ***N*-way set associative cache** (where N is a number, i.e. 2)
  - Instead of mapping anywhere in the cache, a memory reference can map only to the *subset* of cache slots
  - Similar to direct mapped cache
    - Memory reference maps to a limited number of locations
  - Similar to fully associated cache
    - Memory reference maps to more than one potential location (so we need to search in parallel)

Set Associative Cache

- The number of cache blocks per set can vary
- Example: 2-way set associative cache
  - Each set contains two different memory blocks





Logical view

Linear view

Set Associative Cache

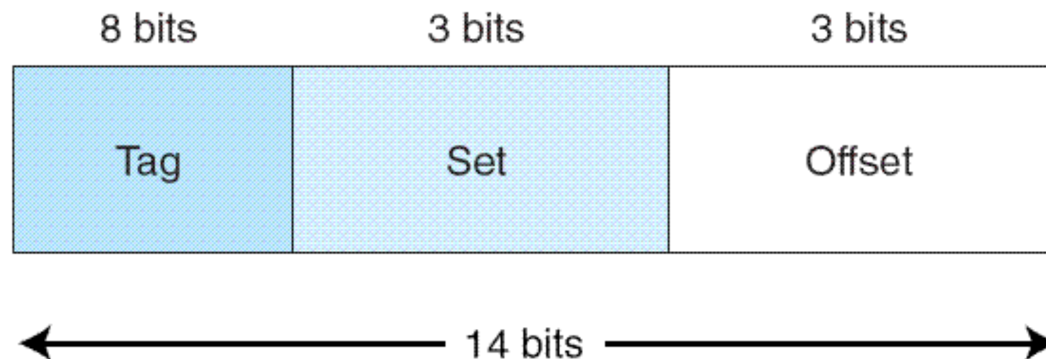
- Memory references are divided into three fields
  - Tag – Uniquely identifies the memory address
  - Set – *New!* Which set does the address map to?
  - Offset – Chooses the word within the cache block

Example 2 – Set Associative Cache

- Memory configuration
  - 2-way set associative cache
  - Word-addressable main memory of 2^{14} words
  - Cache size: 16 blocks
  - Block size: 8 words
- **What do we know about the main memory and cache?**

Example 2 – Set Associative Cache

- **What do we know about the main memory and cache?**
  - Cache has 16 blocks
    - Each set has 2 blocks
    - There are 8 sets in cache
  - Divide up address
    - Set field is 3 bits (2^3 = 8 sets)
    - Offset field is 3 bits (2^3 = 8 words in a block)
    - Tag field is 8 bits (all remaining bits from 14-bit long address)



Example 3 – Set Associative Cache

- Memory configuration
  - 4-way set associative cache
  - 2^{24} words of main memory
  - Cache size: 128 blocks
  - Block size: 8 words
- **How many blocks of main memory are there?**
  - Each block contains 8 (2^3) words
  - 2^{24} words / 2^3 words per block = 2^{21} blocks

Example 3 – Set Associative Cache

- Memory configuration
  - 4-way set associative cache
  - 2^{24} words of main memory
  - Cache size: 128 blocks
  - Block size: 8 words
- **What is the format of a memory address as seen by the cache?**
  - **Offset** field: 3 bits (to specify one of the 8 words in each block)
  - **Set** field: 5 bits (128 total blocks / 4 blocks per set = 32 sets)
  - **Tag** field: 16 bits (remaining bits of 24-bit address)

Example 3 – Set Associative Cache

- Memory configuration
  - 4-way set associative cache
  - 2^{24} words of main memory
  - Cache size: 128 blocks
  - Block size: 8 words
- **To what cache set will address `0x138F29` map?**
  - `0x138F29` = 0001 0011 1000 1111 0010 1001
  - Set field is 00101 = Set 5
    - Any one of the 4 blocks within that set!

Example 4 – Hit Ratio

- Suppose a program loops 2 times, accessing even addresses in memory from 6 to 40 inclusive. Compute the hit ratio for a 2-way set associative cache with a total of 16 two-word blocks
  - Assume all cache contents are initially invalid
  - Assume oldest cache entry is victim

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - 16 blocks / 2 blocks per set = 8 sets (3 bit set field)
  - Each address has a:
    - 3 bit set field
    - 1 bit offset (word) field
  - Observation: Every 16 addresses, the values in these fields repeat

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - **First loop!**
  - Access to address 6 (0110, Set=3, Offset=0)
    - Miss! (Set 3 is invalid)
    - Words 6-7 are loaded into one block (tag = 0)
  - Access to address 8 (1000, Set=4, Offset=0)
    - Miss! (Set 4 is invalid)
    - Words 8-9 are loaded into one block, tag = 0

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - Same for addresses 10, 12, 14, 16, 18 and 20 (6 misses to sets 5, 6, 7, 0, 1 and 2)
  - Access to address 22 (10110, Set=3, Offset=0)
    - Miss!
    - One block in the cache is invalid
    - The other block has tag 0 (for words 6-7)
    - Words 22-23 are loaded into the other block, tag = 1

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - Same for addresses 24 to 36
    (7 misses to sets 4, 5, 6, 7, 0, 1 and 2)
  - Access to address 38 (100110, Set=3, Offset=0)
    - Miss!
    - Tag (2) doesn't match either block
      (words 6-7 and 22-23)
    - Words 38-39 (tag=2) replace words 6-7 (tag=0)
  - Same thing for address 40
    Miss to set 4, words 40-41 replace words 8-9, tag=2

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - Same thing for address 40
    Miss to set 4, words 40-41 replace words 8-9, tag=2
  - Total for the first loop:
    - 18 misses
    - 0 hits
    - Hit ratio is 0%

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - **Second loop!**
  - Access to address 6 is a miss to set 3
    - Words 6-7 replace 22-23, tag = 0
  - Access to address 8 is a miss to set 4
    - Words 8-9 replace 24-25, tag = 0
  - Access to address 10 is a hit to set 5
    - Finally!
  - Accesses to addresses 12 through 20 are hits (5 total)

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - Access to address 22 is a miss to set 3
    - Words 22-23 replace 38-39, tag = 1
  - Access to address 24 is a miss to set 4
    - Words 24-25 replace 40-41, tag = 1
  - Accesses to addresses 26 though 36 are hits (6 total)
  - Access to address 38 is a miss to set 3
    - Words 38-39 replace 6-7, tag = 2

Example 4 – Hit Ratio

- 2-way set associative cache with 16 two-word blocks
  - Access to address 40 is a miss to set 4
    - Words 40-41 replace 8-81, tag = 2
  - Totals for the second loop:
    - 6 misses
    - 12 hits
    - Hit ratio is 66.66%
  - Total for entire program:
    - 24 misses, 12 hits
    - **Hit ratio is = 33.33%**

Cache Replacement Policies

Replacement Policy

- In a fully associative or set associative cache, a **replacement policy** ("algorithm") is run whenever we need to evict a block from cache
- **What would the perfect replacement policy be?**
  - Look into the future to see which blocks won't be needed for the longest period of time – evict those first!
  - This is often called the "*oracle*", as in a prophet…
- The perfect replacement policy is **impossible to implement (unless you have a time machine)**, but it serves as a benchmark to compare actual implementable algorithms against

Replacement Policy

- Algorithm 1
  - **Least recently used (LRU)**
  - Keeps track of the last time that a block was assessed in the cache
  - Evict the block that has been unused for the longest period of time
- **Drawbacks?**
  - Complexity! RU has to maintain an access history for each block, which ultimately slows down the cache

Replacement Policy

- Algorithm 2
  - **First-in, first-out (FIFO)**
  - The block that has been in the cache the longest is evicted, regardless of when it was last used
- **Strengths and weaknesses?**
  - Strengths – Easier to implement
  - Weaknesses – The oldest block in the cache might be the most popular!
    - *If we evict it and it is popular, we'll get it back in the cache soon enough…*

Replacement Policy

- Algorithm 3
  - **Random replacement**
  - Picks a block at random and replaces it with a new block
- **Strengths and weaknesses?**
  - Strengths – Simple to implement. Never thrashes
  - Weaknesses – Might evict a block that will be needed often or needed soon

Cache and Writing

- Up to now, we have talked about reading from main memory
  - And getting faster reads via the cache!
- What about writing to main memory?
  - **Can we get faster writes with a cache?**
- Yes! We can write data not to main memory, but to the (faster) cache instead!

Cache and Writing

- Writing to the cache poses a problem, though
  - If the cache block has been modified from what is in memory, we can't just evict it when we need space – it must be written back to memory first
  - New term – "**Dirty" blocks**
    - Blocks that have been updated while they were in the cache but not written back to main memory yet
  - Cache replacement policies must take into account dirty blocks when deciding who (and how) to evict from the cache

Cache and Writing

Write Through

- Updates cache and main memory simultaneously on every write
- Pro – Simple!
- Con – slows down the access time on updates
  - Usually negligible because the majority of accesses tend to be reads, not writes

Write Back

- Updates memory only when the block is selected for replacement
- Pro – memory traffic is minimized
- Con – The value in memory does not always agree with the value in cache (causing problems in multi-core / multi-processor systems with many caches)

Memory Access Time

Effective Access Time

- The performance of hierarchical memory is measured by its effective access time (EAT)
- EAT is a weighted average
  - Takes into account the hit ratio and relative access times of successive levels of memory
- EAT for a two-level memory:
  - **EAT = H × AccessC + (1-H) × AccessMM**
    - H is the cache hit rate
    - AccessC and AccessMM are the access times for cache and main memory, respectively
  - This equation can be extended to any number of memory levels

Effective Access Time

- Example computer system
  - Main memory access time: 200ns
  - Cache access time: 10ns
  - Cache hit rate: 99%
- Suppose access to cache and main memory occurs **concurrently** (i.e. the accesses overlap)
- EAT = 0.99(10ns) + 0.01(200ns)
  = 9.9ns + 2ns
  = **11.9ns**

Effective Access Time

- Example computer system
  - Main memory access time: 200ns
  - Cache access time: 10ns
  - Cache hit rate: 99%
- Suppose access to cache and main memory occurs **sequentially** (i.e. the accesses do not overlap)
- EAT = 0.99(10ns) + 0.01(10ns + 200ns)
  = 9.9ns + 2.1ns
  = **12ns**

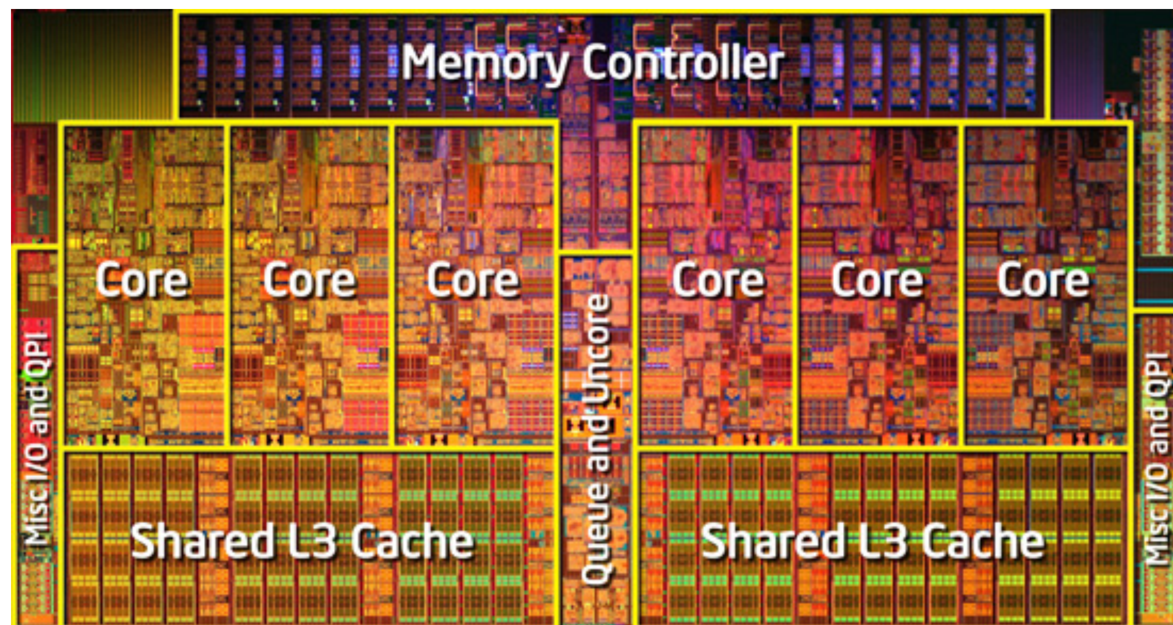
Cache Variations

Cache Variations

- Many variations on cache designs
- **Unified cache** – both instructions and data are cached together
- **Harvard cache** – separate caches for data and instructions
  - Provides better locality (i.e. performance) but increases complexity
  - Can get a similar benefit by simply providing a larger unified cache

Cache Example – Intel Core i7 980x

- High-end 6 core processor with a sophisticated multi-level cache hierarchy
- 3.5GHz, 1.17 billion transistors (!!!)



Cache Example – Intel Core i7 980x

- Each processor core has its own a L1 and L2 cache
  - 32kB Level 1 (L1) data cache
    - 8-way set associative, 64 byte block (“line”) size
  - 32kB Level 1 (L1) instruction cache
    - 4-way set associative, 64-byte block size
  - 256kB Level 2 (L2) cache (both instruction and data)
    - 8-way set associative, 64-byte block size
- The entire chip (all 6 cores) **share** a single 12MB Level 3 (L3) cache
  - 16-way set associative, 64-byte block size

Cache Example – Intel Core i7 980x

- Access time? (Measured in 3.5GHz clock cycles)
  - 4 cycles to access L1 cache
  - 9-10 cycles to access L2 cache
  - 48 cycles to access L3 cache
- Smaller caches are faster to search
  - And can also fit closer to the processor core
- Larger caches are slower to search
  - Plus we have to place them further away

Cache Example – Intel Core i7 980x

- The Intel cache hierarchy is **inclusive**
  - All data in a smaller cache also exists at the next higher level
- Other vendors (e.g. AMD) have **exclusive** caches
  - Only 1 copy of the data in any cache (i.e. if it's in the L1 cache, it cannot also be in the L2 or L3 cache)
- Tradeoffs?
  - Circuit complexity
  - Wasted cache memory space