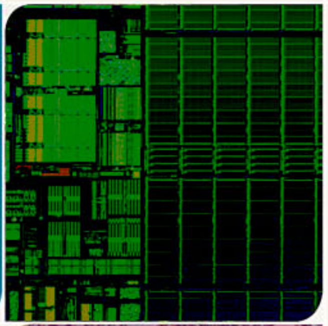
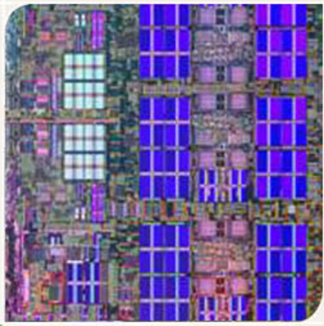
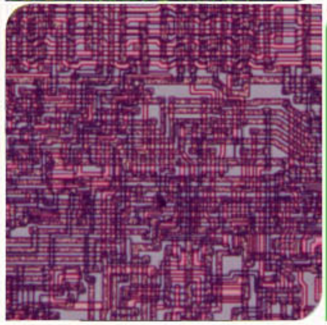
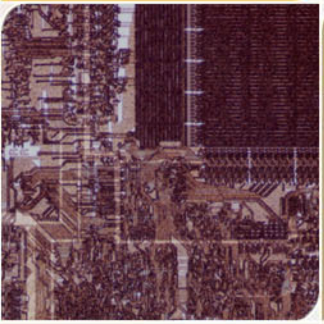





CS4290/CS6290

Fall 2011

Prof. Hyesoon Kim



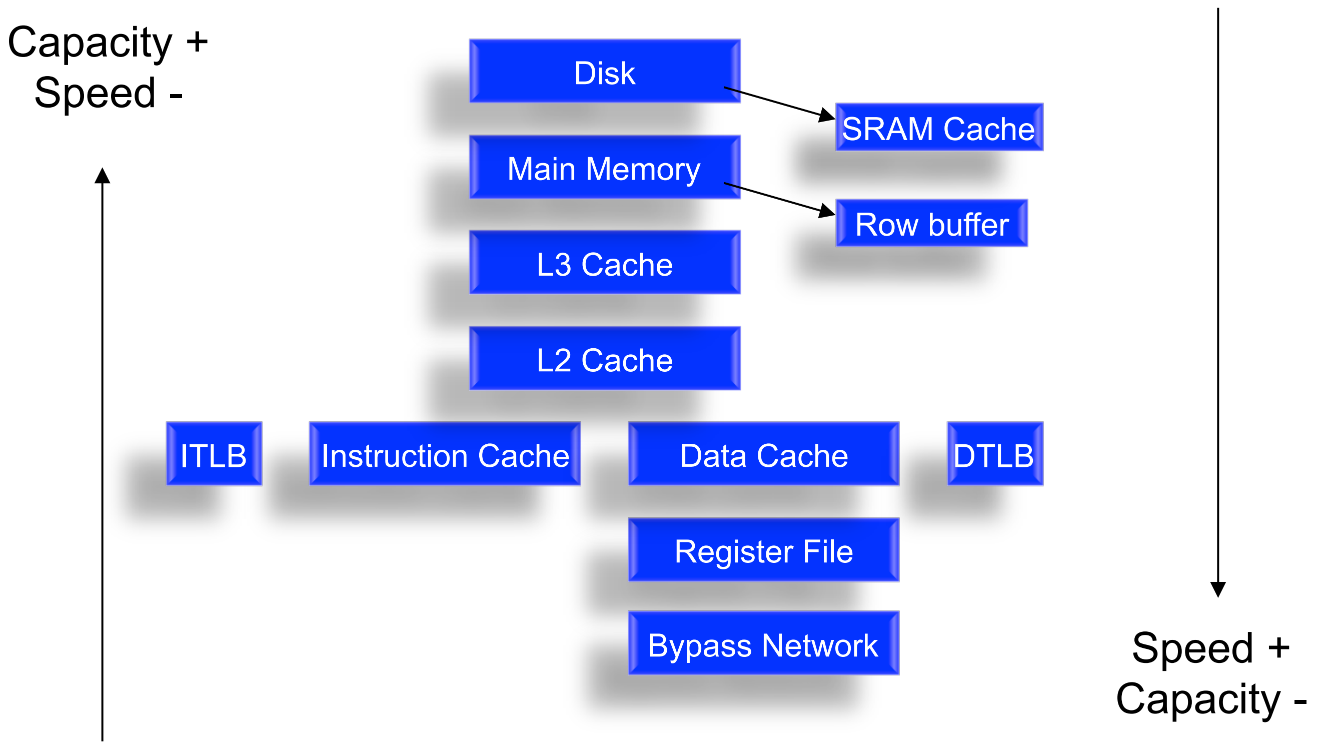
Georgia Tech College of Computing

Thanks to Prof. Loh & Prof. Prvulovic

Locality and Caches

- Data Locality
  - Temporal: if data item needed now, it is likely to be needed again in near future
  - Spatial: if data item needed now, nearby data likely to be needed in near future
- Exploiting Locality: Caches
  - Keep recently used data in fast memory close to the processor
  - Also bring nearby data there

Storage Hierarchy and Locality



Memory Latency is *Long*

- 60-100ns not uncommon
- Quick back-of-the-envelope calculation:
  - 2GHz CPU
  - → 0.5ns / cycle
  - 100ns memory → 200 cycle memory latency!
- Solution: Caches

Cache Basics

- Fast (but small) memory close to processor
- When data referenced
  - If in cache, use cache instead of memor
  - If not in cache, bring into cache (actually, bring entire **block** of data, too)
  - Maybe have to kick something else out to do it!
- Important decisions
  - Placement: where in the cache can a block go?
  - Identification: how do we find a block in cache?
  - Replacement: what to kick out to make room in cache?
  - Write policy: What do we do about stores?

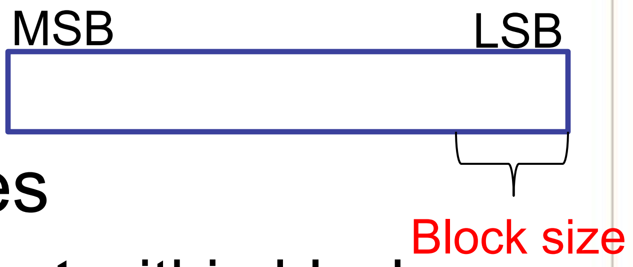
Key: Optimize the *average* memory access latency

Review questions

- Memory addresses A, A+1, A+2, A+3, A+4
  - Spatial locality or temporal locality?:
  - Spatial locality
- Memory addresses A, B,C, A,B,C,A,B,C
  - Spatial locality or temporal locality?
  - Temporal locality

Cache Basics

- Cache consists of block-sized **lines**
  - Line size typically power of two
  - Typically 16 to 128 bytes in size
- Example
  - Suppose block size is 128 bytes
    - Lowest seven bits determine **offset** within block
  - Read data at address A=0x7fffa3f4
  - Address begins to block with **base** address 0x7fffa380



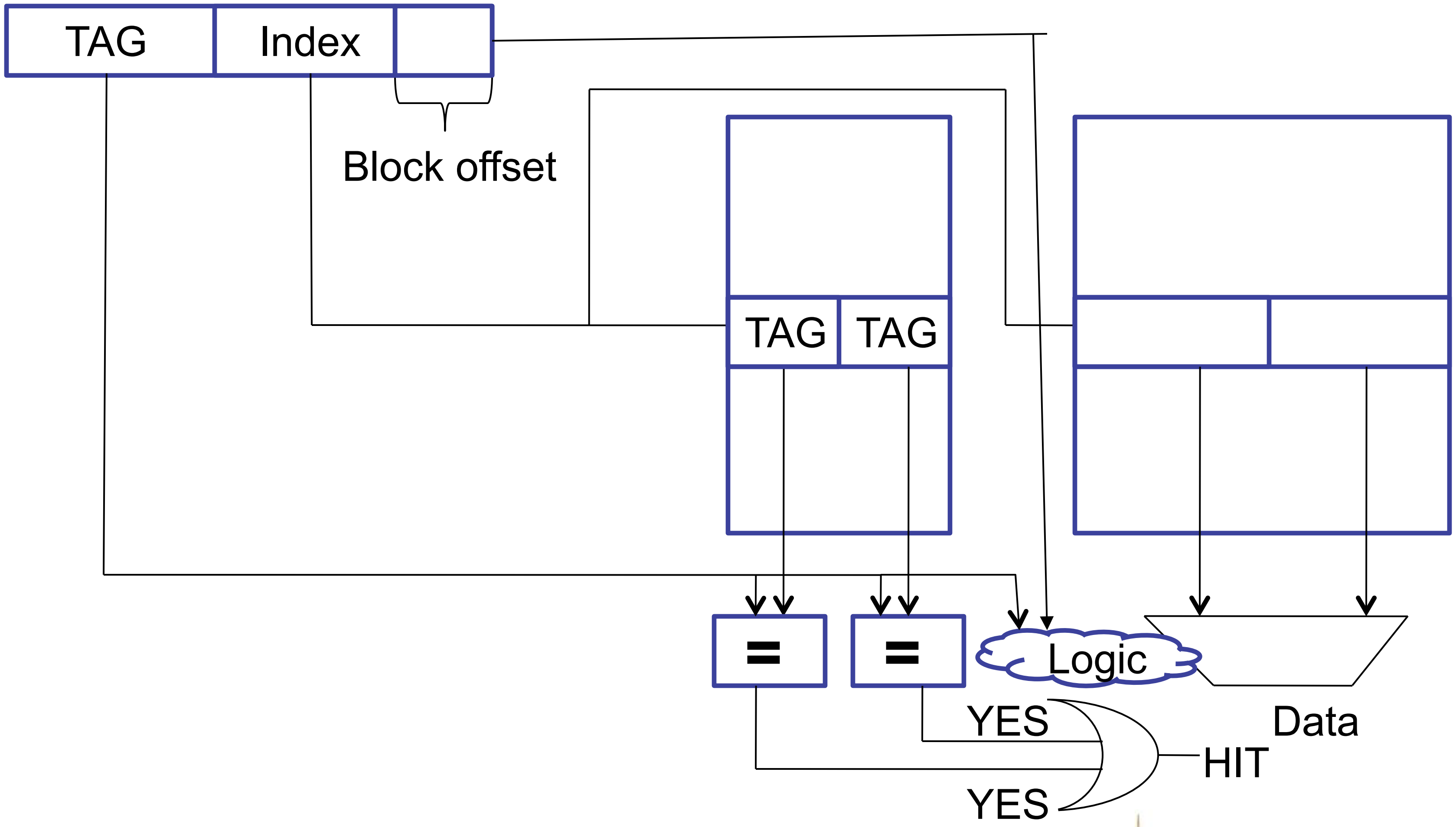
Cache Placement

- Placement
  - Which memory blocks are allowed into which cache lines
- Placement Policies
  - Direct mapped (block can go to only one line)
  - Fully Associative (block can go to any line)
  - Set-associative (block can go to one of N lines)
    - E.g., if N=4, the cache is 4-way set associative
    - Other two policies are extremes of this (E.g., if N=1 we get a direct-mapped cache)

Cache Identification

- When address referenced, need to
  - Find whether its data is in the cache
  - If it is, find where in the cache
  - This is called a cache **lookup**
- Each cache line must have
  - A **valid** bit (1 if line has data, 0 if line empty)
    - We also say the cache line is valid or invalid
  - A **tag** to identify which block is in the line (if line is valid)

Index



Parallel/Serial TAG and Data Array Access

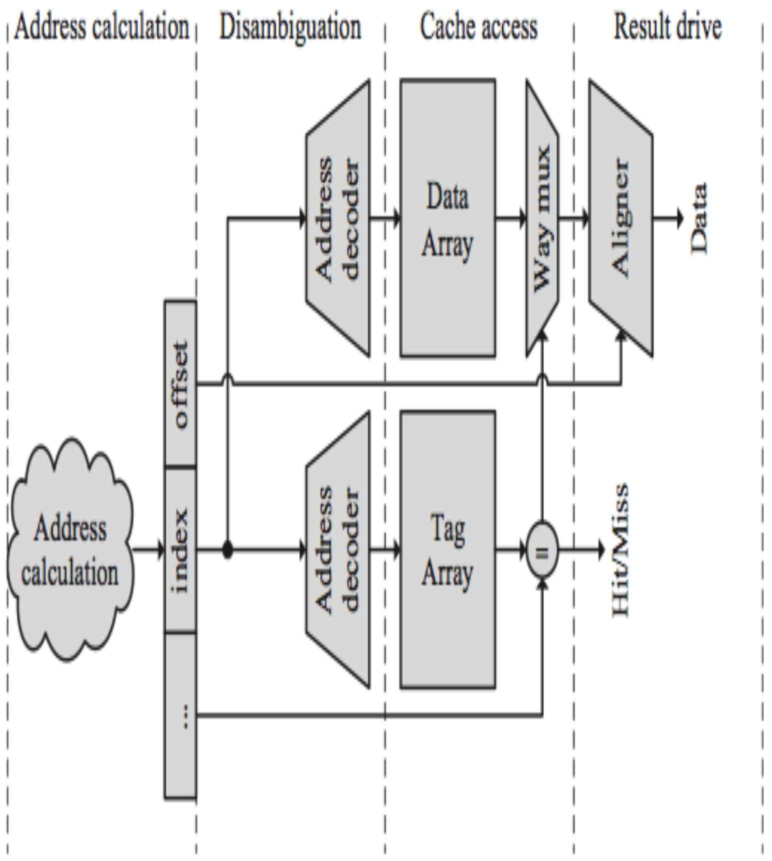


FIGURE 2.2: Parallel tag and data array access pipeline.

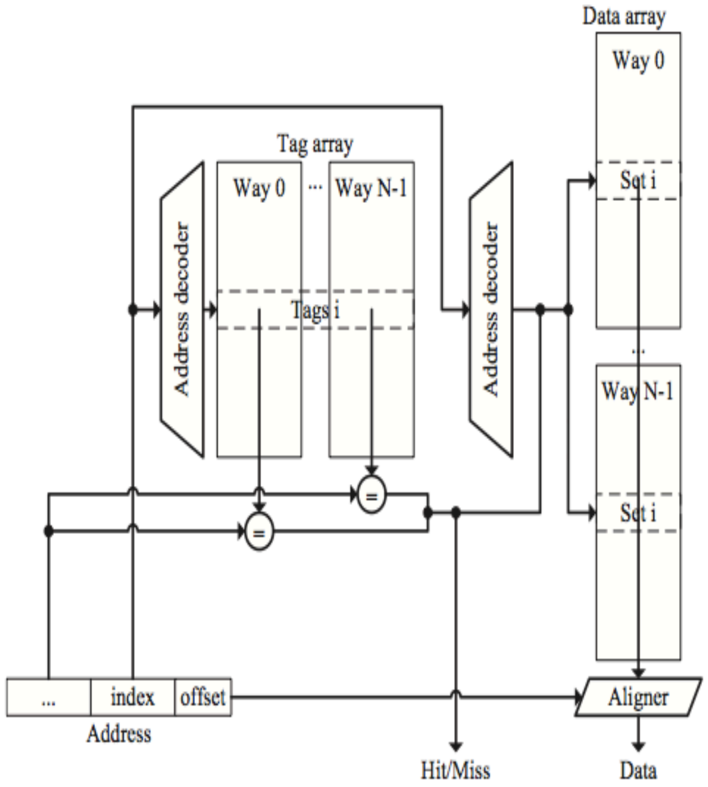


FIGURE 2.3: High-level logical cache organization with serial tag and data array access.

Synthesis lecture series: microarchitecture

Cache Replacement

- Need a free line to insert new block
  - Which block should we kick out?
- Several strategies
  - Random (randomly selected line)
  - FIFO (line that has been in cache the longest)
  - LRU (least recently used line)
  - LRU Approximations (Pseudo LRU)

Implementing LRU

- Have LRU counter for each line in a set
- When line accessed
  - Get old value X of its counter
  - Set its counter to max value
  - For every other line in the set
    - If counter larger than X, decrement it
- When replacement needed
  - Select line whose counter is 0

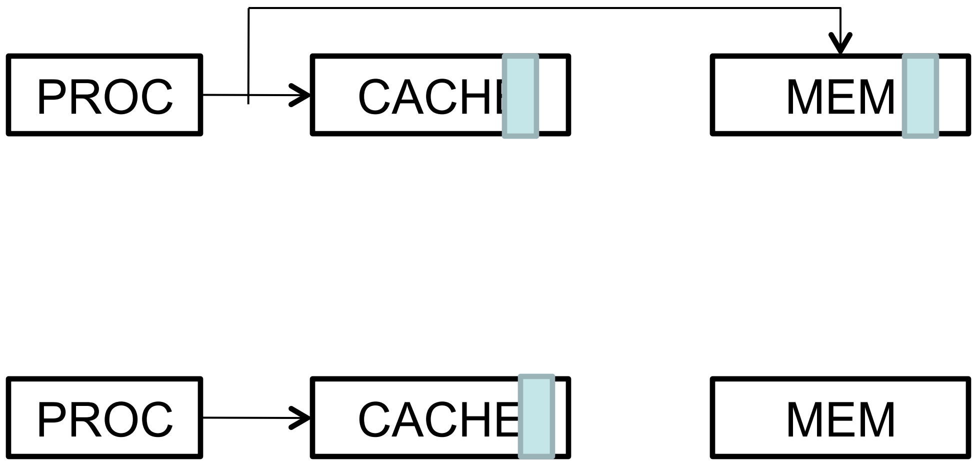
Review questions-II

- Here is a series of address references given as word address: 1,4,8,5,20,17,19,56,9,11,4,4 3,5,6,9,17. Assuming a direct-mapped cache with 16 one-word blocks that is initially empty, label each reference in the list as a hit or miss and show the final contents of the cache.

Write Policy

- Do we allocate cache lines on a write?
  - Write-allocate
    - A write miss brings block into cache
  - No-write-allocate
    - A write miss leaves cache as it was
  - Pros and cons?
    - Depends on temporal locality
- Do we update memory on writes?
  - Write-through
    - Memory immediately updated on each write
  - Write-back
    - Memory updated when line replaced

Write Through/Write Back



Write-through

Write-back

replacement

Write-Back Caches

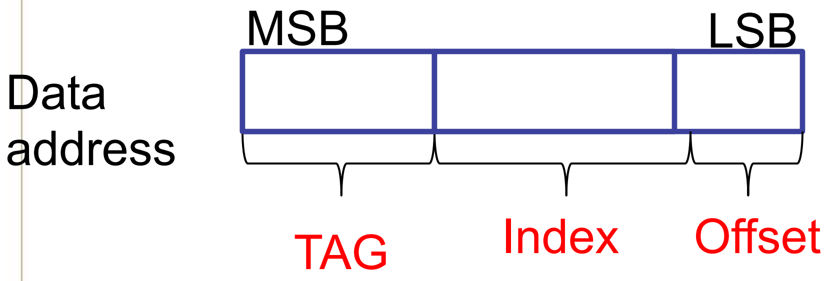
- Need a ***Dirty*** bit for each line (stored in the Tag!)
  - A dirty line has more recent data than memory
- Line starts as ***clean*** (not dirty)
- Line becomes dirty on first write to it
  - Memory not updated yet, cache has the only up-to-date copy of data for a dirty line
- Replacing a dirty line
  - Must write data back to memory (write-back)

Tag Storage

- Any information related to cache other than data is stored in the tag storage.
- Not only tag bits, information for replacement, dirty bits (if we need), valid bit

(in the future, cache coherence state information)

TAG



- Offset: Byte offset in block
- Index: Which set in the cache is the block located
- Tag: need to match address tag in cache
- Set: Group of blocks corresponding to same index

Cache Performance

- Miss rate
  - Fraction of memory accesses that miss in cache
  - Hit rate = 1 – miss rate
- Average memory access time

  AMAT = hit time + miss rate * miss penalty
- Memory stall cycles

$$\text{CPUtime} = \text{CycleTime} \times (\text{Cycles}_{\text{Exec}} + \text{Cycles}_{\text{MemoryStall}})$$

$$\text{Cycles}_{\text{MemoryStall}} = \text{CacheMisses} \times (\text{MissLatency}_{\text{Total}} - \text{MissLatency}_{\text{Overlapped}})$$

Improving Cache Performance

- AMAT = hit time + miss rate * miss penalty
  - Reduce miss penalty
  - Reduce miss rate
  - Reduce hit time
- $\text{Cycles}_{\text{MemoryStall}}$ = CacheMisses x ($\text{MissLatency}_{\text{Total}}$ – $\text{MissLatency}_{\text{Overlapped}}$)
  - Increase overlapped miss latency
  - Increase memory level parallelism

Improving Cache Performance (2)

- Memory latency = 100 cycles
- 16KB cache, 3 cycle latency, 85% hit rate
- What's the baseline AMAT?

= 3+100*0.15 = 18

- What's the best way to reduce latency?
  - Smaller 8KB cache: 1 cycle latency, 75% hit rate

  = 1+ 100*0.25 = 26
  - Larger 32KB cache: 4 cycle latency, 90% hit rate

  = 4 + 100*0.1 = 14

Reducing Cache Miss Penalty (1)

- Multilevel caches
  - Very Fast, small Level 1 (L1) cache
  - Fast, not so small Level 2 (L2) cache
  - May also have slower, large L3 cache, etc.
- Why does this help?
  - Miss in L1 cache can hit in L2 cache, etc.

$\text{AMAT} = \text{HitTime}_{L1} + \text{MissRate}_{L1}\text{MissPenalty}_{L1}$

$\text{MissPenalty}_{L1} = \text{HitTime}_{L2} + \text{MissRate}_{L2}\text{MissPenalty}_{L2}$

$\text{MissPenalty}_{L2} = \text{HitTime}_{L3} + \text{MissRate}_{L3}\text{MissPenalty}_{L3}$

Multi-Level Caches

- Memory latency = 100 cycles
- 16KB L1 cache, 3 cycle latency, 85% hit rate
- Use two levels of caching
  - Smaller 8KB cache: 1 cycle latency, 75% hit rate
  - Larger 128KB cache: 6 cycle latency, 60% hit rate
- Exclusion Property
  - If block is in L1 cache, it is *never* in L2 cache
  - Saves some L2 space
- Inclusion Property
  - If block A is in L1 cache, it *must* also be in L2 cache

Reducing Cache Miss Penalty (2)

- Early Restart & Critical Word First
  - Block transfer takes time (bus too narrow)
  - Give data to loads before entire block arrive
- Early restart
  - When needed word arrives, let processor use it
  - Then continue block transfer to fill cache line
- Critical Word First
  - Transfer loaded word first, then the rest of block (with wrap-around to get the entire block)
  - Use with early restart to let processor go ASAP

Reducing Cache Miss Penalty (3)

- Increase Load Miss Priority
  - Loads can have dependent instructions
  - If a load misses and a store needs to go to memory, let the load miss go first
  - Need a write buffer to remember stores
- Merging Write Buffer
  - If multiple write misses to the same block, combine them in the write buffer
  - Use block-write instead of a many small writes

Kinds of Cache Misses

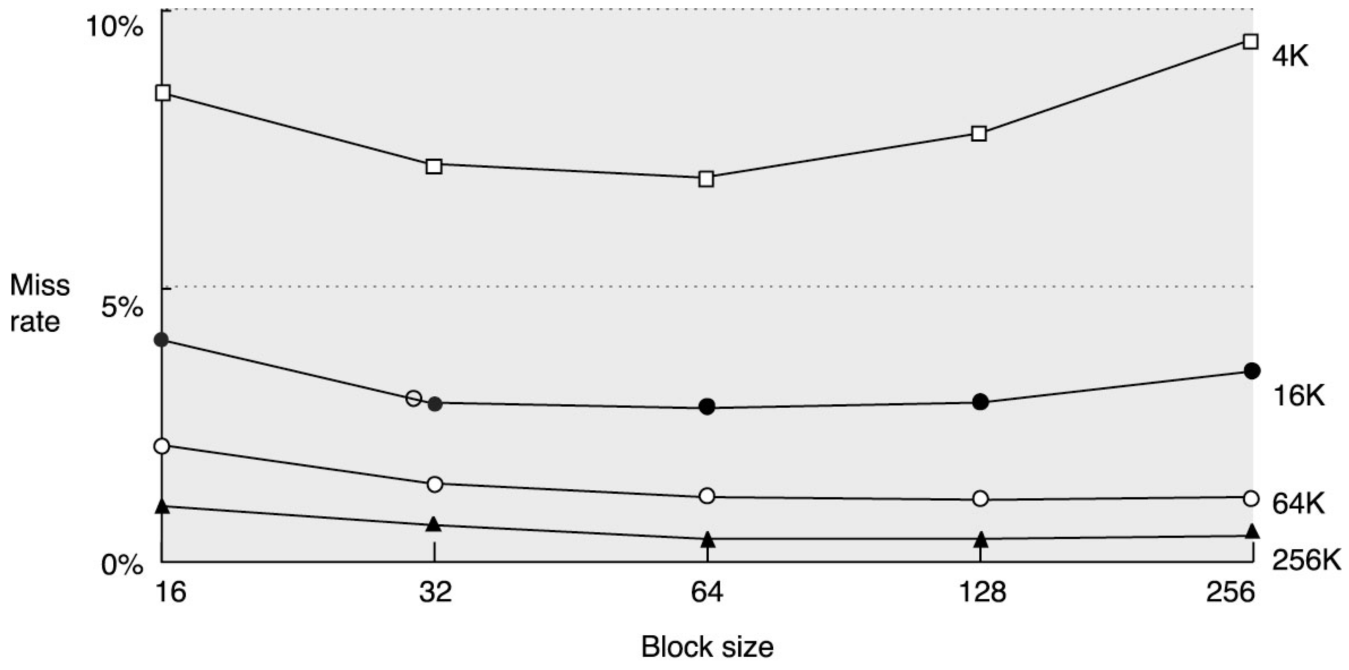
- The "3 Cs"
  - ***Compulsory***: have to have these
    - Miss the first time each block is accessed
  - ***Capacity***: due to limited cache capacity
    - Would not have them if cache size was infinite
  - ***Conflict***: due to limited associativity
    - Would not have them if cache was fully associative

Reducing Cache Miss Rate

- Victim Caches
  - Recently kicked-out blocks kept in small cache
  - If we miss on those blocks, can get them fast
  - Why does it work: [redacted] misses
    - Misses that we have in our N-way set-assoc cache, but would not have if the cache was fully associative
  - Example: direct-mapped L1 cache and a 16-line fully associative victim cache

Reducing Cache Miss Rate (2)

- Larger blocks
  - Helps if there is more spatial locality



Reducing Cache Miss Rate (3)

- Larger caches
  - Fewer capacity misses, but longer hit latency!
- Higher associativity
  - Fewer conflict misses, but longer hit latency!

  - … need to work through AMAT equations to figure out which is better

Reducing Cache Miss Rate (4)

- Pseudo Associative Caches
  - Similar to way prediction
  - Start with direct mapped cache
  - If miss on "primary" entry, try another entry
  - Results in varying access time
- Compiler optimizations
  - Loop interchange
  - Blocking (e.g., tiled matrix multiplication)

Loop Interchange

- For loops over multi-dimensional arrays
  - Example: matrices (2-dim arrays)
- Change order of iteration to match layout
  - Gets better spatial locality
  - Layout in C: last index changes first

```
for(j=0;j<10000;j++)
  for(i=0;i<40000;i++)
    c[i][j]=a[i][j]+b[i][j];
```

⇨

`a[i][j]` and `a[i+1][j]`
are 10000 elements apart

Blocking (tiling)

Reduce the working set size.

To make the code fit into the cache

Example

1D case

```
  for (k = 0; k < N; k++)
      for (j=0; j<M; j++)
      if (B[j] == a[k]) sum[k]++;
```

If one element in B is 4B and the total cache size is 1KB.

1KB/4B = 256 elements; So

```
for (j=0; j<M/256; j++)
 for (k = 0; k < N; k++)
         for (l = 0; l< 256; l++)
         if (B[j*256+l] == a[k]) sum[k]++;
```

Review

- Improving cache performance
  - Cache block sizes?
  - Cache write polices?
  - Multi-level caches
  - Inclusion/exclusion property
  - Write buffers
  - Pseudo associativity cache, way predictor
  - Victim caches
  - Early restart & Critical words first

Other Software Optimization for Caches

- Data alignment
  - malloc
- Use Hint bits:
  - Indicate temporal locality
  - Dead block

Question

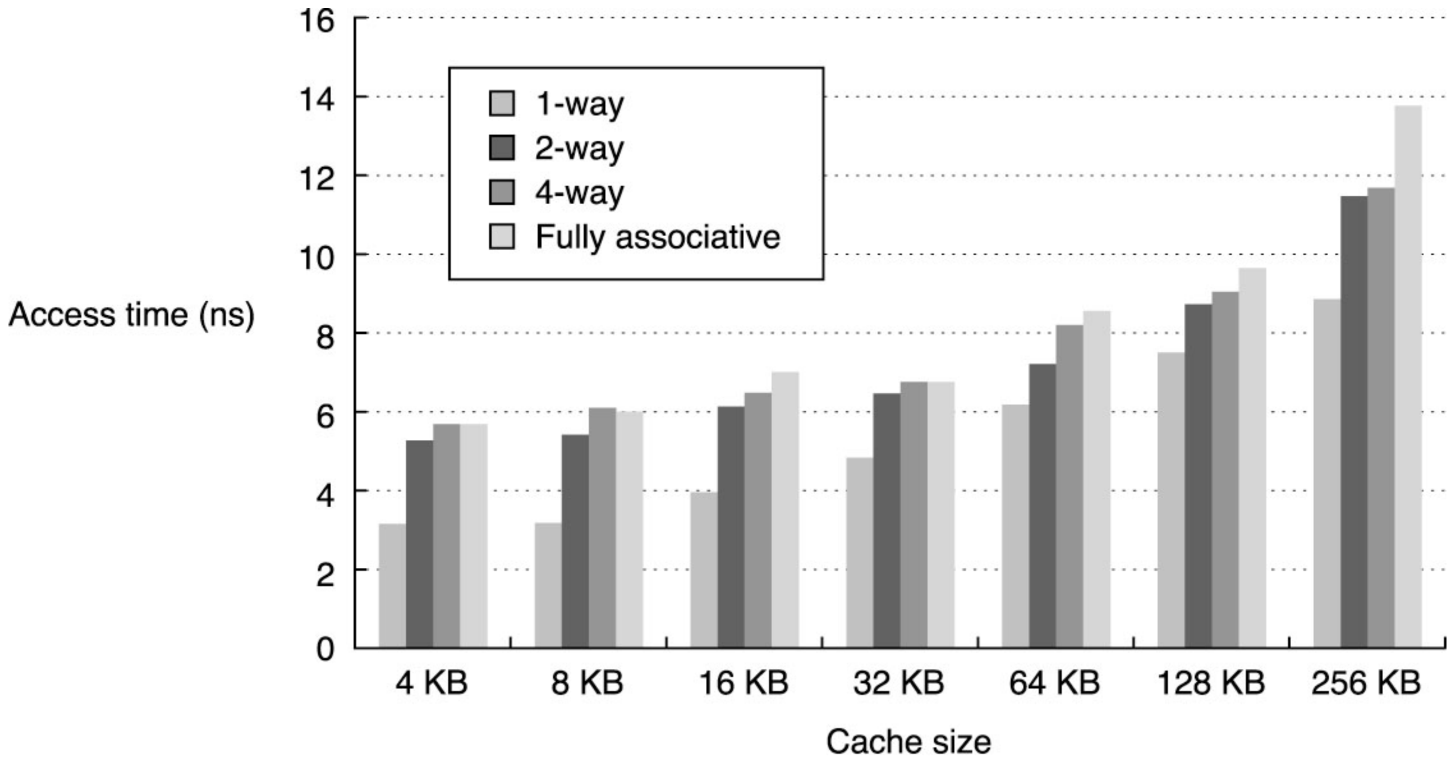
```
#define MAX_LAST_NAME_SIZE 16
typedef struct  _TAGPHONE_BOOK_ENTRY {
        char LastName[MAX_LAST_NAME_SIZE];
        char FirstNAME[16]; char email[16]; char phone[10]; char cell[10]; char addr1[16];
        char addr2[16]; char city[16]; char state[2]; char zip[5];
        _TAGPHONE_BOOK_ENTRY *pNext;
        } Phone BOOK
        PhoneBook *FindName(char Last[], PhoneBook *pHead)
        {
        While (pHead != NULL) {
         if (stricmp (Last, pHead->LastName) == 0)
        Return pHead;
        pHead = pHead->pNext;
        }
        return NULL;
        }
```

What are the problems and how to improve the cache locality?
In Software solutions

Solutions

Reducing Hit Time (1)

- Small & Simple Caches are faster



Reducing Hit Time (2)

- Avoid address translation on cache hits
- Software uses virtual addresses, memory accessed using physical addresses
  - virtual memory
- HW must translate virtual to physical
  - Normally the first thing we do
  - Caches accessed using physical address
  - Wait for translation before cache lookup
- Idea: index cache using virtual address
  - Virtual index physical tag (later lecture)

Reducing Hit Time (3)

- Pipelined Caches
  - Improves bandwidth, but not latency
  - Essential for L1 caches at high frequency
    - Even small caches have 2-3 cycle latency at N GHz
  - Also used in many L2 caches
- Trace Caches
  - For instruction caches
- Way Prediction
  - Speeds up set-associative caches
  - Predict which of N ways has our data, fast access as direct-mapped cache
  - If mispredicted, access again as set-assoc cache

Hiding Miss Latency

- Idea: overlap miss latency with useful work
  - Also called “latency hiding”
- Non-blocking (Lockup-free) caches
  - A blocking cache services one access at a time
    - While miss serviced, other accesses blocked (wait)
  - Non-blocking caches remove this limitation
    - While miss serviced, can process other requests
- Prefetching
  - Predict what will be needed and get it ahead of time

Non-Blocking (Lockup-Free) Caches

- Hit Under Miss
  - Allow cache hits while one miss in progress
  - But another miss has to wait
- Miss Under Miss, Hit Under Multiple Misses
  - Allow hits and misses when other misses in progress
  - Memory system must allow multiple pending requests

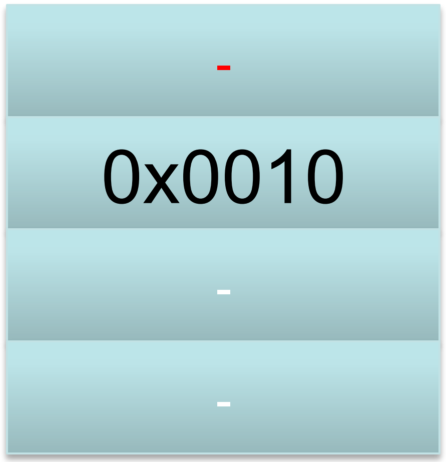
- MSHR (Miss Information/Status Holding Register): Stores unresolved miss information for each miss that will be handled concurrently.

MSHR

- Track address, data, and status for multiple outstanding cache misses
- Provide correct memory ordering, other information (e.g, cache coherence)
- Valid flag,
- Write bits
- # of entries: # of outstanding memory requests
- aka MOB (Memory ordering buffer)

Example

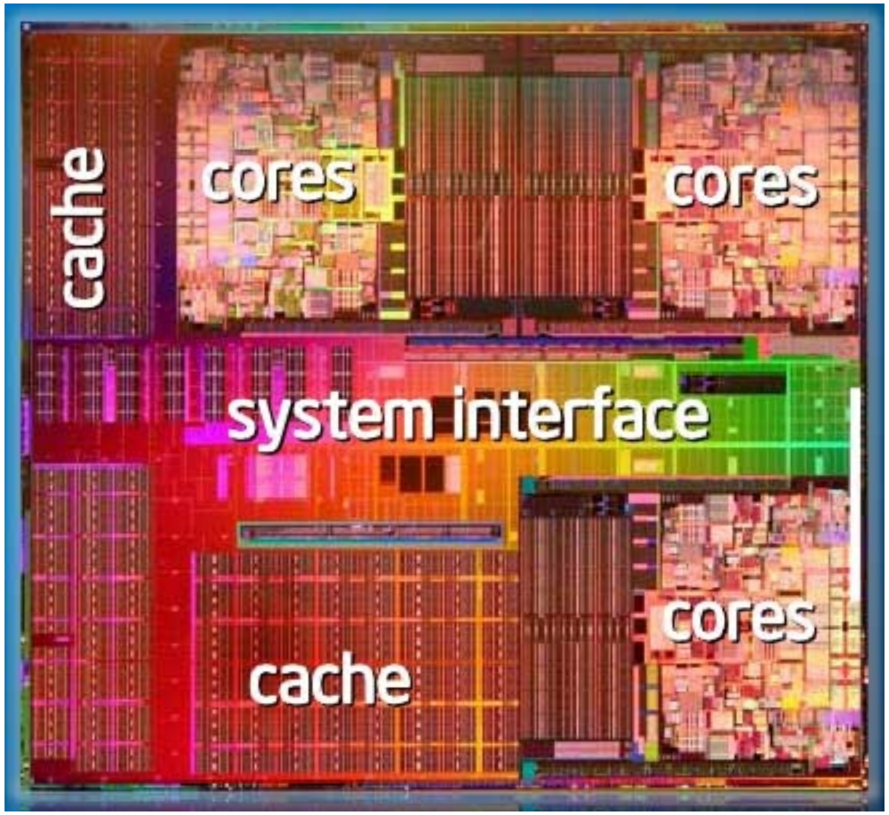
- Cache block size 64B
- Inst1: LD 0x0001
- inst2: LD 0x0010
- inst3: LD 0x0005
- Cache hit: 2 cycles, cache miss: 30 cycles



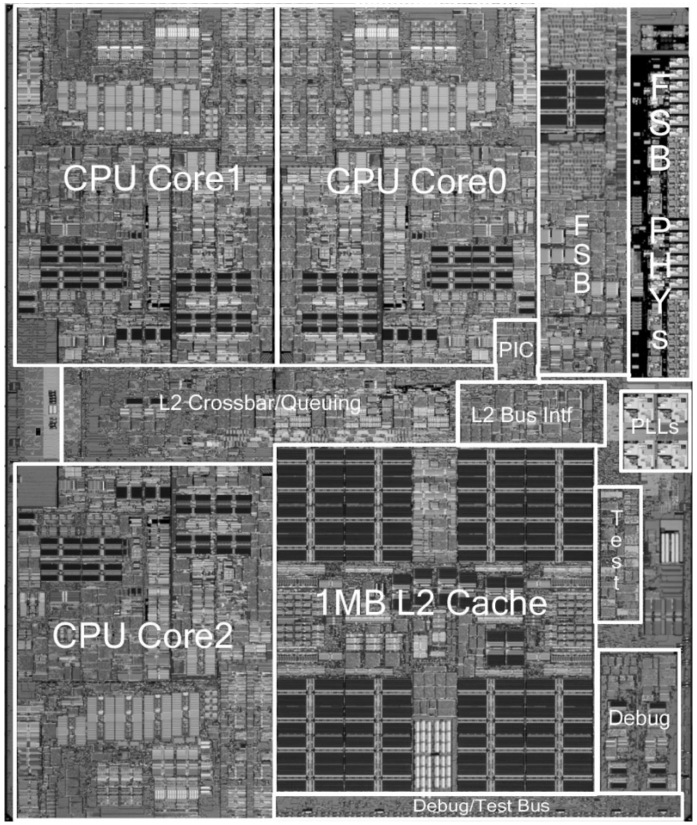
blocking cache: 30 + 2 + 2 + inst3 execution time
Non-blocking cache 30 + inst 3 execution time

Another question inst 1: cycle 1, inst 2: cycle 10, inst 3: cycle 15, when
Will inst 3 be ready?

Today's caches



Intel's chip (Dunnington)16MB L3 cache



XBox

Georgia Tech College of Computing

FAQ Lab #2

- Where an op should wait if there is data dependence?
  - Wait at the last latch in the ID stage. So basically do not enter EX stage until all source operands are ready.
- N-stage deeper pipeline means?
  - Pipelined structure (see FAQ in the homepage)
- Do we need pc addresses?