Computer Organization and Architecture

Chapter 4

Cache Memory

Note: Appendix 4A will not be covered in class, but the material is interesting reading and may be used in some homework problems.

Characteristics of Memory Systems

Location	**Performance**
Processor	Access time
Internal (main)	Cycle time
External (secondary)	Transfer rate
Capacity	**Physical Type**
Word size	Semiconductor
Number of words	Magnetic
Unit of Transfer	Optical
Word	Magneto-Optical
Block	**Physical Characteristics**
Access Method	Volatile/nonvolatile
Sequential	Erasable/nonerasable
Direct	**Organization**
Random	
Associative	

Location

- CPU
  - Registers and control unit memory
- Internal
  - Main memory and cache
- External
  - Storage devices (paper tape, cards, tapes, discs, flash cards, etc.)

Capacity

- Word size
  - The natural unit of organisation
  - Typically number of bits used to represent an integer in the processor
- Number of words
  - Most memory sizes are now expressed in bytes
  - Most modern processors have byte-addressable memory but some have word addressable memory
  - Memory capacity for A address lines is 2^A addressable units

Unit of Transfer

- Internal
  - Usually governed by data bus width
- External
  - Usually a block which is much larger than a word (typical disk 512 - 4096 bytes)
- Addressable unit
  - Smallest location which can be uniquely addressed
  - Some systems have only word addressable memory while many have byte addressable memory
  - A block or even cluster of blocks on most disks

Access Methods (1)

- Sequential
  - Start at the beginning and read through in order
  - Access time depends on location of data and previous location
  - e.g. tape
- Direct
  - Individual blocks have unique address
  - Access is by jumping to vicinity plus sequential search
  - Access time depends on location and previous location
  - e.g. disk

Access Methods (2)

- Random
  - Individual addresses identify locations exactly
  - Access time is independent of location or previous access
  - e.g. RAM
- Associative
  - Data is located by a comparison with contents of a portion of the store
  - Access time is independent of location or previous access
  - All memory is checked simultaneously; access time is constant
  - e.g. cache

Performance

- From user's perspective the most important characteristics of memory are capacity and performance
- Three performance parameters:
  - Access time
  - Cycle Time
  - Transfer Rate
- Access time (latency)
  - For RAM access time is the time between presenting an address to memory and getting the data on the bus
  - For other memories the largest component is positioning the read/write mechanism

Performance

- Cycle Time
  - Primarily applied to RAM; access time + additional time before a second access can start
  - Function of memory components and system bus, not the processor
- Transfer Rate – the rate at which data can be transferred into or out of a memory unit
  - For RAM TR = 1 / (cycle time)

Transfer rate for other memories

- $T_n = T_a + (n/r)$ where
- T_n =Average time to read or write N bits
- T_a =Average access time
- n = number of bits
- r = transfer rate in bits / second

Physical Types of Memory

- Semiconductor
  - RAM (volatile or non-volatile)
- Magnetic Surface Memory
  - Disk & Tape
- Optical
  - CD & DVD
- Others
  - Magneto-optical
  - Bubble
  - Hologram

Physical Characteristics

- Volatility
  - Does the memory retain data in the absence of electrical power?
- Decay
  - Ranges from tiny fractions of a second (volatile DRAM) to many years (CDs, DVDs)
- Erasable
  - Can the memory be rewritten? If so, how fast? How many erase cycles can occur?
- Power consumption

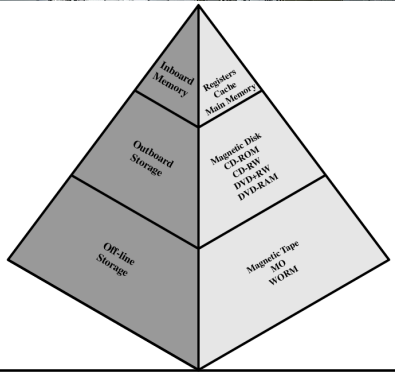
Organization

- Physical arrangement of bits into words
- Not always obvious, e.g., interleaved memory (examples later)

Memory Hierarchy

- For any memory:
  - How fast?
  - How much?
  - How expensive?
- Faster memory => greater cost per bit
- Greater capacity => smaller cost / bit
- Greater capacity => slower access
- Going down the hierarchy:
  - Decreasing cost / bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access by processor

Memory Hierarchy - Diagram



Memory Hierarchy

- Registers
  - In CPU
- Internal or Main memory
  - May include one or more levels of cache
  - "RAM"
- External memory
  - Backing store

Hierarchy List

- Registers
- L1 Cache
- L2 Cache
- Main memory
- Disk cache
- Magnetic Disk
- Optical
- Tape
- (and we could mention punch cards, etc at the very bottom)

Locality of Reference

- Two or more levels of memory can be used to produce average access time approaching the highest level
- The reason that this works well is called "locality of reference"
- In practice memory references (both instructions and data) tend to cluster
  - Instructions: iterative loops and repetitive subroutine calls
  - Data: tables, arrays, etc. Memory references cluster in short run

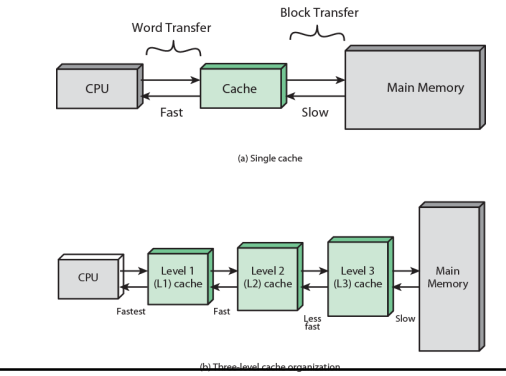
Cache

- A small amount of fast memory that sits between normal main memory and CPU
- May be located on CPU chip or module
- Intended to allow access speed approaching register speed
- When processor attempts to read a word from memory, cache is checked first

Cache Memory Principles

- If data sought is not present in cache, a block of memory of fixed size is read into the cache
- Locality of reference makes it likely that other words in the same block will be accessed soon

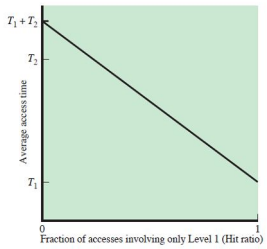
Cache and Main Memory



A Simple two-level cache

- Level 1: 1000 words, 0.01μs
- Level 2: 100,000 words 0.1μs
- If word in L1 processor has direct access else word copied from L2 into L1
- Av Access Time as function of hit ratio H:
  H * 0.01μs + (1-H)* 0.11μs
- With H near 1 access time approaches 0.01μs

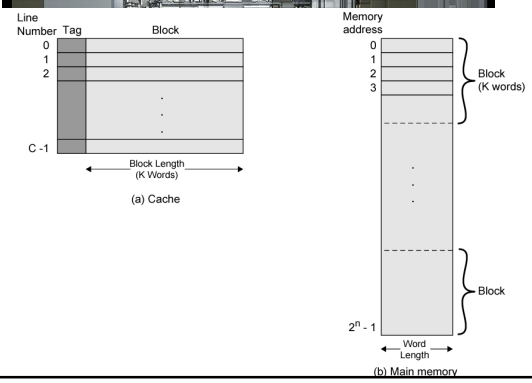
Two-level cache performance



Two-level disk access

- Principles of two-level memories can be applied to disk as well as RAM
- A portion of main memory can be used as a disk cache
  - Allows disk writes to be clustered; largest component of disk access time is seek time
  - Dirty (modified) datamay be requested by the program before it is even written back to disk

Cache/Main Memory Structure



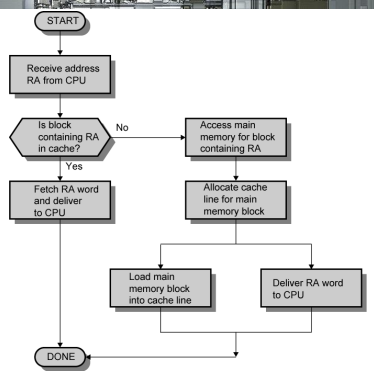
Cache view of memory

- N address lines => 2^n words of memory
- Cache stores fixed length blocks of K words
- Cache views memory as an array of M blocks where $M = 2^n/K$
- A block of memory in cache is referred to as a line. K is the line size
- Cache size of C blocks where C < M (considerably)
- Each line includes a tag that identifies the block being stored
- Tag is usually upper portion of memory address

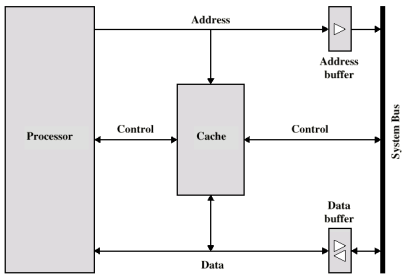
Cache operation – overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Cache Read Operation - Flowchart



Typical Cache Organization



Cache organization

- The preceding diagram illustrates a shared connection between the processor, the cache and the system bus (look-aside cache)
- Another way to organize this system is to interpose the cache between the processor and the system bus for all lines (look-through cache)

Elements of Cache Design

- Addresses (logical or physical)
- Size
- Mapping Function (direct, assoociative, set associative)
- Replacement Algorithm (LRU, LFU, FIFO, random)
- Write Policy (write through, write back, write once)
- Line Size
- Number of Caches (how many levels, unified or split)

Note that cache design for High Performance Computing (HPC) is very different from cache design for other computers

Some HPC applications perform poorly with typical cache designs

Cache Size does matter

- Cost
  - More cache is expensive
  - Would like cost/bit to approach cost of main memory
- Speed
  - But we want speed to approach cache speed for all memory access
  - More cache is faster (up to a point)
  - Checking cache for data takes time
  - Larger caches are slower to operate

Comparison of Cache Sizes

Processor	Type	Year of Introduction	L1 Cache[a]	L2 cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128 to 256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256 to 512 KB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 kB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 kB/32 kB	4 MB	—
Itanium 2	PC/server	2002	32 kB	256 KB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24-48 MB

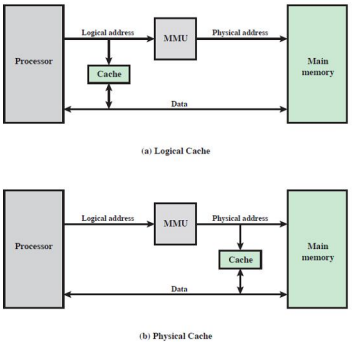
Virtual Memory

- Almost all modern processors support *virtual memory* (Ch 8)
- Virtual memory allows a program to treat its memory space as single contiguous block that may be considerably larger than main memory
- A memory management unit takes care of the mapping between virtual and physical addresses

Logical Cache

- A logical (virtual) cache stores virtual addresses rather than physical addresses
- Processor addresses cache directly without going through MMU
- Obvious advantage is that addresses do not have to be translated by the MMU
- A not-so-obvious disadvantage is that all processes have the same virtual address space – a block of memory starting at 0
  - The same virtual address in two processes usually refers to different physical addresses
  - So either flush cache with every context switch or add extra bits

Logical and Physical Cache



Look-aside and Look-through

- Look-aside cache is parallel with main memory
- Cache and main memory both see the bus cycle
  - Cache hit: processor loaded from cache, bus cycle terminates
  - Cache miss: processor AND cache loaded from memory in parallel
- Pro: less expensive, better response to cache miss
- Con: Processor cannot access cache while another bus master accesses memory

Look-through cache

- Cache checked first when processor requests data from memory
  - Hit: data loaded from cache
  - Miss: cache loaded from memory, then processor loaded from cache
- Pro:
  - Processor can run on cache while another bus master uses the bus
- Con:
  - More expensive than look-aside, cache misses slower

Mapping Function

- There are fewer cache lines than memory blocks so we need
  - An algorithm for mapping memory into cache lines
  - A means to determine which memory block is in which cache line
- Example elements:
  - Cache of 64kByte
  - Cache block of 4 bytes
    - i.e. cache is 16k (2^{14}) lines of 4 bytes
  - 16MBytes main memory
  - 24 bit address (2^{24}=16M)

(note: Pentium cache line = 32 bytes until Pentium 4 (128 bytes))

Direct Mapping

- Each block of main memory maps to only one cache line
  - i.e. if a block is in cache, it must be in one specific place
- Mapping function is `i = j modulo m` (`i = j % m`) where
  i = cache line number
  j = main memory block number
  m = number of cache lines
- Address is in two parts
- Least Significant w bits identify unique word
- Most Significant s bits specify one memory block
- The MSBs are split into a cache line field r and a tag of s-r bits (most significant)

Direct Mapping Address Structure

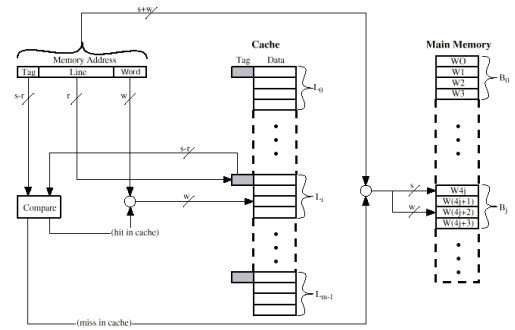
Tag s-r	Line or Slot r	Word w
8	14	2

- 24 bit address, 2 bit word identifier (4 byte block)
- 22 bit block identifier (s)
  - 8 bit tag (=22-14) and 14 bit slot or line
  - Example: AB1402 tag=AB line=0500 word=2
    Note: 1402 = 0001 0100 0000 0010
    Remove l.s. 2 bits = 0001 0100 0000 00 = 00 0101 0000 0000 = 0500
- There are 2^s blocks in memory
- No two blocks with the same line number can have the same Tag field
  - AC1400, 041403, C71401 …
- Check contents of cache by finding line and checking Tag
  - Line is 0500 for all of these
  - If mem request is AB1402 tag at 0500 must = AB

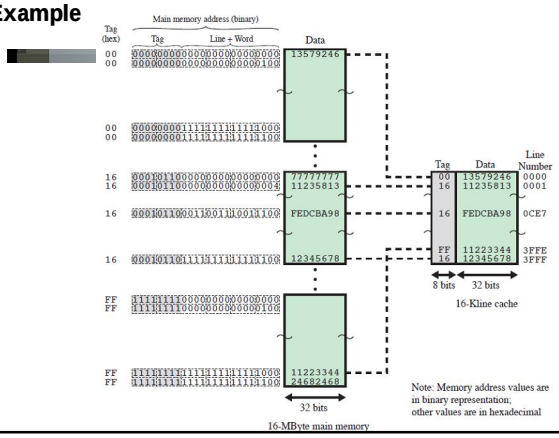
Direct Mapping

- Parking lot analogy: think of the cache as a parking lot, with spaces numbered 0000-9999
- With a 9 digit student id, we could assign parking spaces based on the middle 4 digits: xxx PPPP yy
- Easy to find your parking space
- Problem if another student is already there!
- Note that with memory addresses, the *middle* bits are used as a line number
  - Locality of reference suggests that memory references close in time will have the same high-order bits

Direct Mapping Cache Organization



Example



Direct Mapping Summary

- Address length = (s + w) bits where w = $\log_2$(block size)
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory
  = $2^{s+w}/2^w = 2^s$
- Size of line field is r bits
  - Number of lines in cache = m = 2^r
  - Size of tag = (s – r) bits
- Size of cache 2^{r+w} bytes or words

Direct Mapping Cache Line Table

Cache line	Main Memory blocks held
0	0, m, 2m, 3m…2^s-m
1	1,m+1, 2m+1…2^s-m+1
…	
m-1	m-1, 2m-1,3m-1…2^s-1

0	000000,010000,…,FF0000
1	000004,010004,…,FF0004
…	
m-1	00FFFC,01FFFC,…,FFFFFC

Direct Mapping Pros & Cons

- Pro
  - Simple
  - Inexpensive
- Con
  - Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high (thrashing)
- Victim cache
  - A solution to direct mapped cache thrashing
  - Discarded lines are stored in a small “victim” cache (4 to 16 lines)
  - Victim cache is fully associative and resides between L1 and next level of memory

Associative Mapping

- A main memory block can load into any line of cache
- Memory address is interpreted as 2 fields: tag and word
- Tag uniquely identifies block of memory
- Every line’s tag is examined simultaneously for a match
  - Cache searching gets expensive because a comparator must be wired to each tag
  - A comparator consists of XNOR gates (true when both inputs are true)
  - Complexity of comparator circuits makes fully associative cache expensive

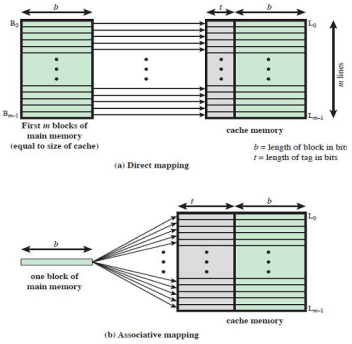
XNOR , this is the logical complement of the exclusive OR (XOR) gate

INPUT		OUTPUT
A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Associative Mapping

- Because no bit field in the address specifies a line number the cache size is not determined by the address size
- Associative-mapped memory is also called "content-addressable memory."
- Items are found not by their address but by their content
  - Used extensively in routers and other network devices
  - Corresponds to associative arrays in Perl and other languages
- Primary disadvantage is the cost of circuitry

Direct Mapping compared to Associative

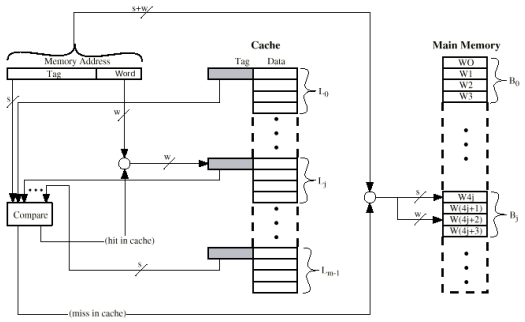


Associative Mapping Address Structure

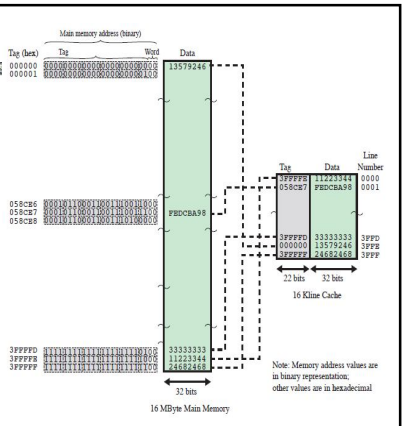
Tag 22 bit	Word 2 bit

- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block
- e.g.
  - Address Tag Data Cache line
  - FFFFFD FFFFFC 24682468 3FFF

Fully Associative Cache Organization



Example



Associative Mapping

- Parking lot analogy: there are more permits than spaces
- Any student can park in any space
- Makes full use of parking lot
  - With direct mapping many spaces may be unfilled
- Note that associative mapping allows flexibility in choice of replacement blocks when cache is full
- Discussed below

Associative Mapping Summary

- Address length = (s + w) bits where w = $\log_2$(block size)
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $2^{s+w}/2^w = 2^s$
- Number of lines in cache = undetermined
- Size of tag = s bits

Set Associative Mapping

- A compromise that provides strengths of both direct and associative approaches
- Cache is divided into a number of sets of lines
- Each set contains a fixed number of lines
- A given block maps to any line in a *given set* determined by that block's address
  - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - 2-way associative mapping
  - A given block can be in one of 2 lines in only one set

Set Associative Mapping

- m = v * k
  - Where m = number of lines in cache, v = number of sets and k = lines/set
  - Lines in cache = sets * lines per set
- i = j modulo v
  - Where I = set number and j = main memory block number
  - Set number = block number % number of sets
- This is referred to as a "k-way" set associative mapping
- Block B_i can be mapped only into lines of set j.

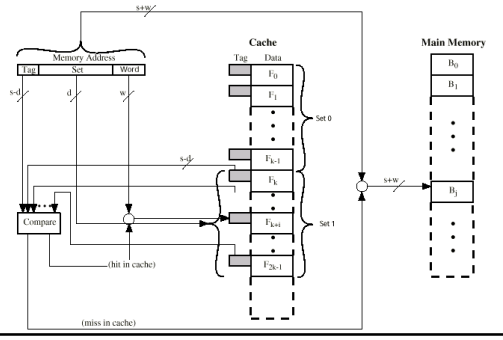
Set Associative Mapping: Parking Analogy

- If we have 10,000 parking spaces we can divide them into 1000 sets of 10 spaces each
- Still use middle digits of id to find your parking place *set:* xxx PPP yyy
- You have a choice of any place in your set
- Our parking lots actually work like this, but the sets are fairly large: Fac/Staff; Commuter; Resident; Visitor

Set Associative Mapping Example

- Assume 13 bit set number
- Block number in main memory is modulo 2^{13} (0010 0000 0000 0000 = 2000h
- 000000, 002000, 004000, … map to same set

K-Way Set Associative Cache Organization

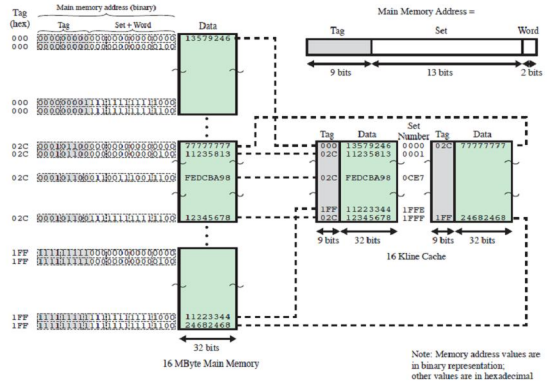


Set Associative Mapping Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit

- Cache control logic sees address as three fields: tag, set and word
- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- e.g
  - Address Tag Data Set number
  - 1FF 7FFC 1FF 12345678 1FFF
  - 001 7FFC 001 11223344 1FFF
- Tags are much smaller than fully associative memories and comparators for simultaneous lookup are much less expensive

Example



Set Associative Mapping Summary

- For a *k*-way set associative cache with *v* sets (each set contains k lines):
  - Address length = (t+d+w) bits where w = $\log_2$(block size) and d = $\log_2$(v)
  - Number of addressable units = 2^{t+d+w} words or bytes
  - Size of tag = t bits
  - Block size = line size = 2^w words or bytes
  - Number of blocks in main memory = 2^{t+d}
  - Number of lines in set = k
  - Number of sets = v = 2^d
  - Number of lines in cache = kv = k * 2^d

Tag (t bits)	Set (d bits)	Word (w bits)

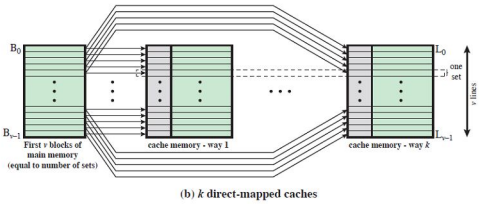
Additional Notes

- Where *v (*# sets) = *m* (# lines in cache) and k = 1 (one line/set) then set associative mapping reduces to direct mapping
- For v=1 (one set) and k=m (# sets = # lines) it reduces to pure associative mapping
- 2 lines/set; v=m/2, k=2 is quite common.
- Significant improvement in hit ratio over direct mapping
- Four-way mapping v=m/4, k=4 provides further modest improvement

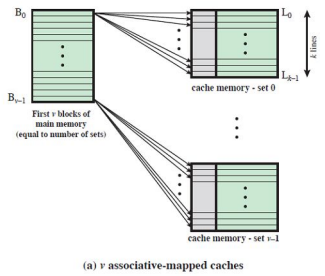
Set Associative Mapping Implementation

- A set associative cache can be implemented as *k* direct mapped caches OR as *v* associative caches
- With k direct mapped caches each direct mapped cache is referred to as a *way*
- The direct mapped implementation is used for small degrees of associativity (small k) and the associative mapped implementation for higher degrees.

Direct mapped implementation

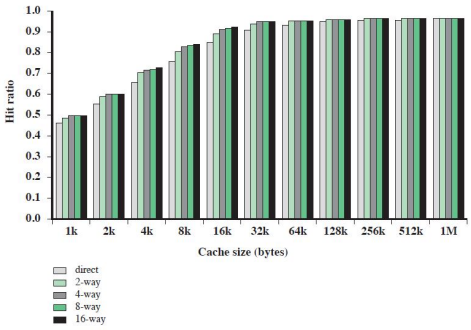


Associative Mapped Implementation



(a) *v* associative-mapped caches

Varying associativity over cache size



Cache replacement algorithms

- When a line is read from memory it replaces some other line already in cache
- Other than direct mapping, there are choices for replacement algorithm
- Any given choice can result in a great speedup for one program and slow-down for some other program
- There is no "best choice" that works for all programs

Replacement Algorithms (1) Direct mapping

- No choice
- Each block only maps to one line
- Replace that line

Replacement Algorithms (2) Associative & Set Associative

- Algorithm is hardware implemented for speed
- Least Recently used (LRU) assumes locality of reference so *most* recently used is likely to be used again
- LRU is easy for 2-way set associative cache
  - Each line has a USE bit
  - When a line is referenced, set USE bit to 1 and set the USE bit for the other line to 0
  - Replace the block whose USE bit is 0
- Implemented in fully-associative caches by keeping list of lines
  - Most recently referenced lines goes to head of list

MRU replacement

- Most Recently Used (MRU) would seem to be an improbable algorithm
- It is useful for some specialized caches where the type of code executing is known (example: database index scan)

Other Algorithms

- First in first out (FIFO)
  - replace block that has been in cache longest
  - Implemented as circular queue
- Least frequently used
  - replace block which has had fewest hits
- Random
  - Almost as good other choices
- LRU is often favored because of ease of hardware implementation

Write Policy

- When a block of memory about to be overwritten in cache:
  - No problem if not modified in cache
  - Has to written back to main memory if modified (dirty)
- Must not overwrite a cache block unless main memory is up to date

Problems with dirty memory

- More than one device may have access to main memory
  - I/O may address main memory directly
  - If word altered in cache, then main memory is invalid
  - If word altered in memory, then cache is invalid
- Multiple CPUs may have individual caches
  - Word altered in one cache may invalidate other caches

Write through

- Simplest technique
- All writes go to main memory as well as cache
- Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date (cache coherency)
- Lots of memory traffic, slows down writes

Write back

- Updates initially made in cache only
- Update bit for cache slot is set when update occurs
- If block is to be replaced, write to main memory only if update bit is set
  - Other caches can get out of sync
- I/O must access main memory through cache
- N.B. Typically 15% of memory references are writes; but can be as high as 50% in some HPC apps

Cache Coherency

- In a bus organization with shared memory and multiple caches coherency has to be maintained between caches as well as cache and memory
- Possible approaches:
  1. Bus watching with write through. Cache controller monitors bus lines and detects writes to memory in cache. Requires write-through policy for ALL cache controllers
  2. Hardware transparency. Extra hardware ensures that a write to one cache updates memory and all other caches
  3. Noncacheable memory. Memory shared between processors is designated as non-cacheable. All accesses to shared memory are cache misses. Mem identified with chip-select logic or high address bits
- More info in Ch. 18

Line Size

- When a cache line is filled it normally includes more than the requested data – some adjacent words are retrieved
- As block size increases, cache hit ratio will also increase because of locality of reference – to a limit
- If block size is too large, possibility of reference to parts of block decreases; there are fewer blocks in cache so more chance of block being overwritten

Line Size

- Relationship between block size and hit ratio is complex and program-dependent
- No optimal formula exists
- General purpose computing uses blocks of 8 to 64 bytes
- In HPC 64 and 128 byte lines are most common

Number of caches: multilevel caches

- With increased logic density caches can be on same chip as processor
- Reduces external bus activity and speeds up execution times
- No bus cycles; shorter data path is faster than 0-wait bus cycles
- Bus is free to do other transfers

Multilevel Caches

- It is usually desirable to have external as well as internal cache
- With only 1 level bus access to memory is slow
- Most contemporary computers have at least 2 levels
  - Internal: Level 1 (L1)
  - External: Level 2 (L2)
- External L2 cache typically built with fast SRAM; uses separate and faster data bus
- Now incorporated on processor chip

L2 and L3 Cache

- Performance improvements depend on hit rates
- Complicates replacement algorithms and write policy
- With L2 cache on-board L3 cache can improve performance just as L2 can improve over L1 alone

Unified and Split Caches

- Split caches have separate caches for instructions and data
  - These tend to be stored in different areas of memory
- Pros of unified cache:
  - Higher rate for given cache size because cache is automatically balanced between instructions and data
  - Only one cache needs to implemented

Split Cache

- Current trend favors split caches
  - Useful for superscalar machines with parallel execution of instructions and prefetching of predicted instructions
  - Split cache eliminates contention for cache between instruction fetch/decode unit and the execution unit (when accessing data)
  - Helps to keep pipeline full because the EU will block the fetch/decode unit otherwise

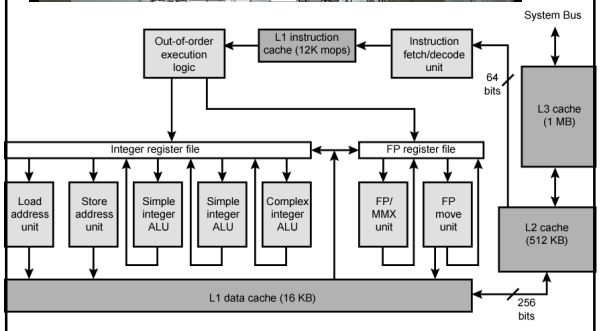
Pentium Cache Evolution

- 80386 – no on chip cache
- 80486 – 8k using 16 byte lines and four way set associative organization
- Pentium (all versions) – two on chip L1 caches
  - Data & instructions
- Pentium III – L3 cache added off chip
- Pentium 4
  - L1 caches
    - 8k bytes
    - 64 byte lines
    - four way set associative
  - L2 cache
    - Feeding both L1 caches
    - 256k
    - 128 byte lines
    - 8 way set associative
  - L3 cache on chip

Pentium Cache Evolution

Problem	Solution	Processor on which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip	Add external L2 cache using faster technology than main memory	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Pentium 4 Block Diagram



Pentium 4 Core Processor

- Fetch/Decode Unit
  - Fetches instructions from L2 cache
  - Decode into micro-ops
  - Store micro-ops in L1 cache
- Out of order execution logic
  - Schedules micro-ops
  - Based on data dependence and resources
  - May speculatively execute
- Execution units
  - Execute micro-ops
  - Data from L1 cache
  - Results in registers
- Memory subsystem
  - L2 cache and systems bus

Pentium 4 Design Reasoning

- Decodes instructions into RISC like micro-ops before L1 cache
- Micro-ops fixed length
  - Superscalar pipelining and scheduling
- Pentium instructions long & complex
- Performance improved by separating decoding from scheduling & pipelining
  - (More later – ch14)
- Data cache is write back
  - Can be configured to write through
- L1 cache controlled by 2 bits in register
  - CD = cache disable
  - NW = not write through
  - 2 instructions to invalidate (flush) cache and write back then invalidate
- L2 and L3 8-way set-associative
  - Line size 128 bytes

Pentium 4 Cache Operating Modes

Control Bits		Operating Mode		
CD	NW	Cache Fills	Write Throughs	Invalidates
0	0	Enabled	Enabled	Enabled
1	0	Disabled	Enabled	Enabled
1	1	Disabled	Disabled	Disabled

Note: CD = 0; NW = 1 is an invalid combination.

ARM Cache Organization

- ARM3 started with 4KB of cache
- ARM design emphasis on few transistors and small, low-power chips has kept cache fairly small

ARM Cache Features

Core	Cache Type	Cache Size (kB)	Cache Line Size (words)	Associativity	Location	Write Buffer Size (words)
ARM720T	Unified	8	4	4-way	Logical	8
ARM920T	Split	16/16 D/I	8	64-way	Logical	16
ARM926EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	16
ARM1022E	Split	16/16 D/I	8	64-way	Logical	16
ARM1026EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	8
Intel StrongARM	Split	16/16 D/I	4	32-way	Logical	32
Intel Xscale	Split	32/32 D/I	8	32-way	Logical	32
ARM1136-JF-S	Split	4-64/4-64 D/I	8	4-way	Physical	32

Write Buffer

- Distinctive feature of ARM cache is a FIFO write buffer between cache and main memory
- When data is written to a bufferable area of memory, data are placed in write buffer at CPU clock speed and CPU continues execution
- Write buffer performs memory write in parallel with processor
- If write buffer is full then CPU is stalled until write buffer drains
- Data from same addresses as write buffer cannot be read until write is complete