

RISC Design:

Memory System Design

Virendra Singh

Associate Professor

Computer **A**rchitecture and **D**ependable **S**ystems **L**ab

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.ee.iitb.ac.in/~viren/>

E-mail: viren@ee.iitb.ac.in

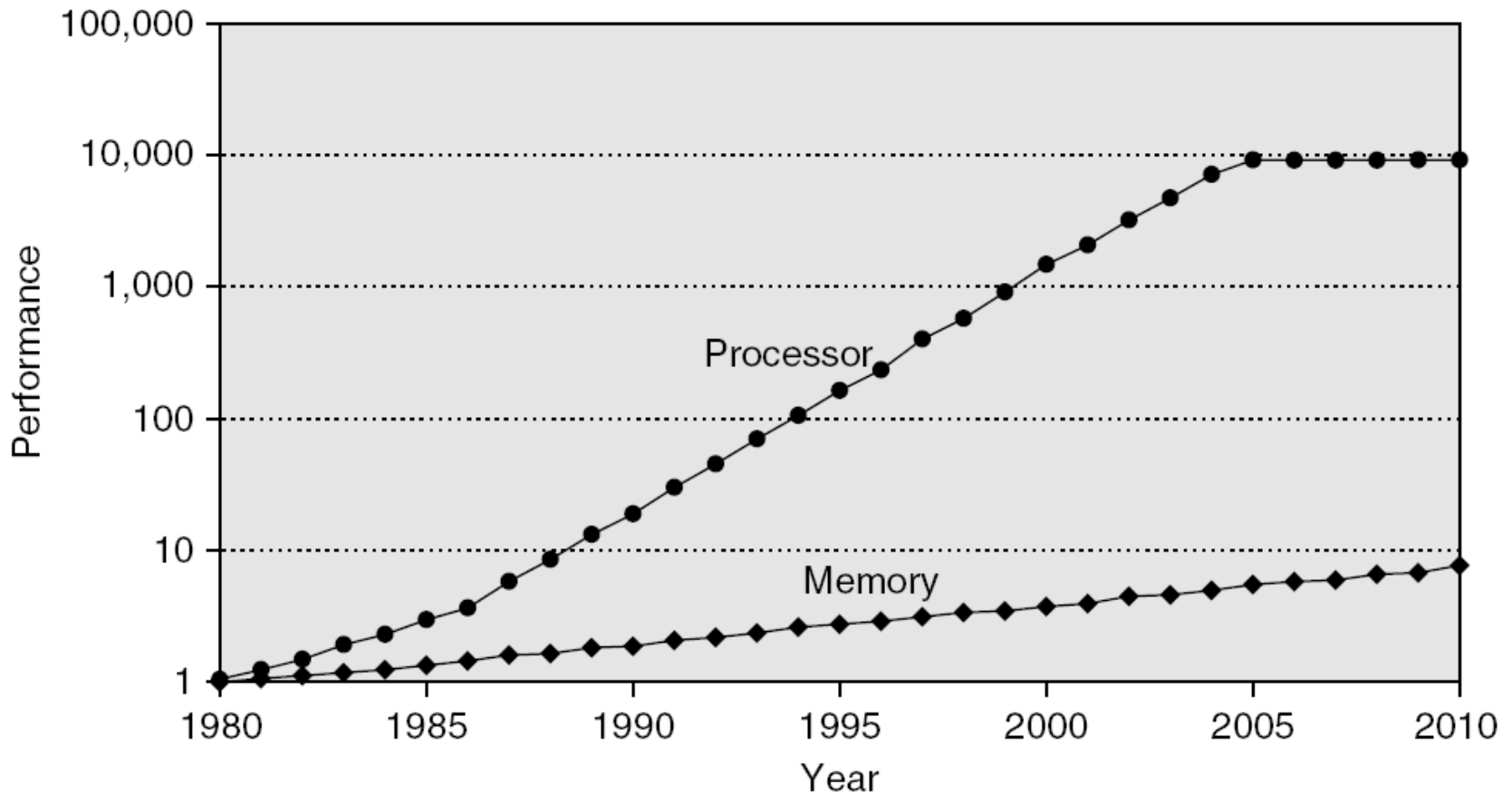
CP-226: Computer Architecture



Lecture 16 (16 March 2013)

CADSL

Memory Performance Gap



Why Memory Hierarchy?

- Need lots of bandwidth

$$BW = \frac{1.0inst}{cycle} \times \left[\frac{1Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4Dref}{inst} \times \frac{4B}{Dref} \right] \times \frac{1Gcycles}{sec}$$
$$= \frac{5.6GB}{sec}$$

- Need lots of storage
 - 64MB (minimum) to multiple TB
- Must be cheap per bit
 - (TB x anything) is a lot of money!
- These requirements seem incompatible

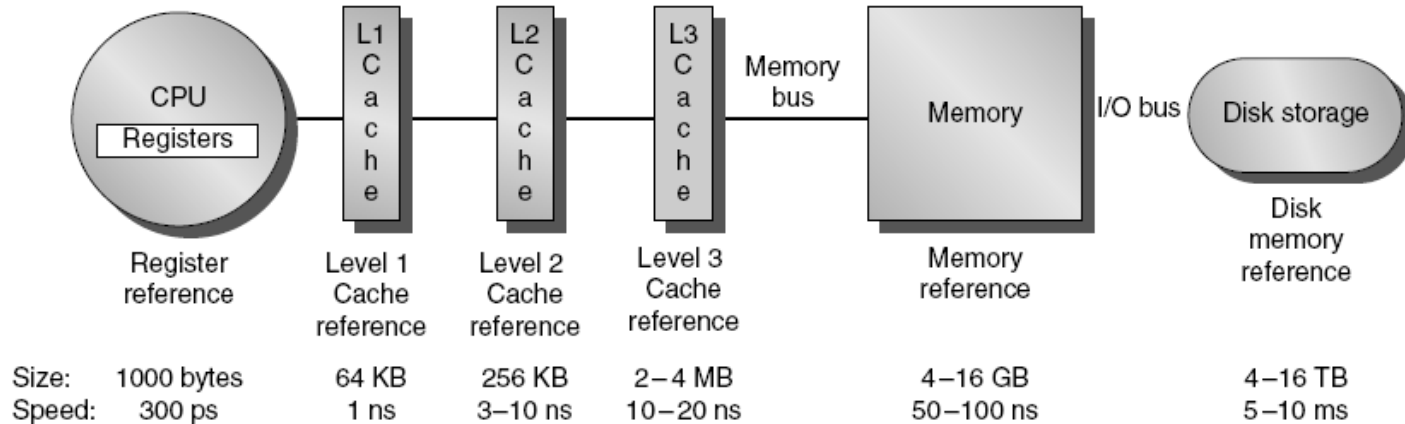


Why Memory Hierarchy?

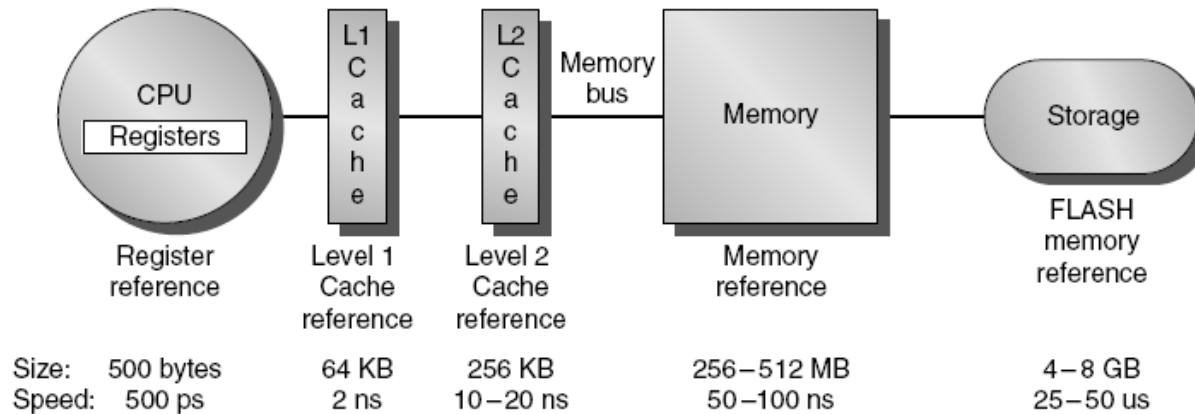
- Fast and small memories
 - Enable quick access (fast cycle time)
 - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
 - Capture larger share of memory
 - Still relatively fast
- Slow huge memories
 - Hold rarely-needed state
 - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory



Memory Hierarchy



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Why Does a Hierarchy Work?

- Locality of reference
 - Temporal locality
 - Reference same memory location repeatedly
 - Spatial locality
 - Reference near neighbors around the same time
- Empirically observed
 - Significant!
 - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set



Why Locality?

- Analogy:
 - Library (Disk)
 - Bookshelf (Main memory)
 - Stack of books on desk (off-chip cache)
 - Opened book on desk (on-chip cache)
- Likelihood of:
 - Referring to same book or chapter again?
 - Probability decays over time
 - Book moves to bottom of stack, then bookshelf, then library
 - Referring to chapter $n+1$ if looking at chapter n ?



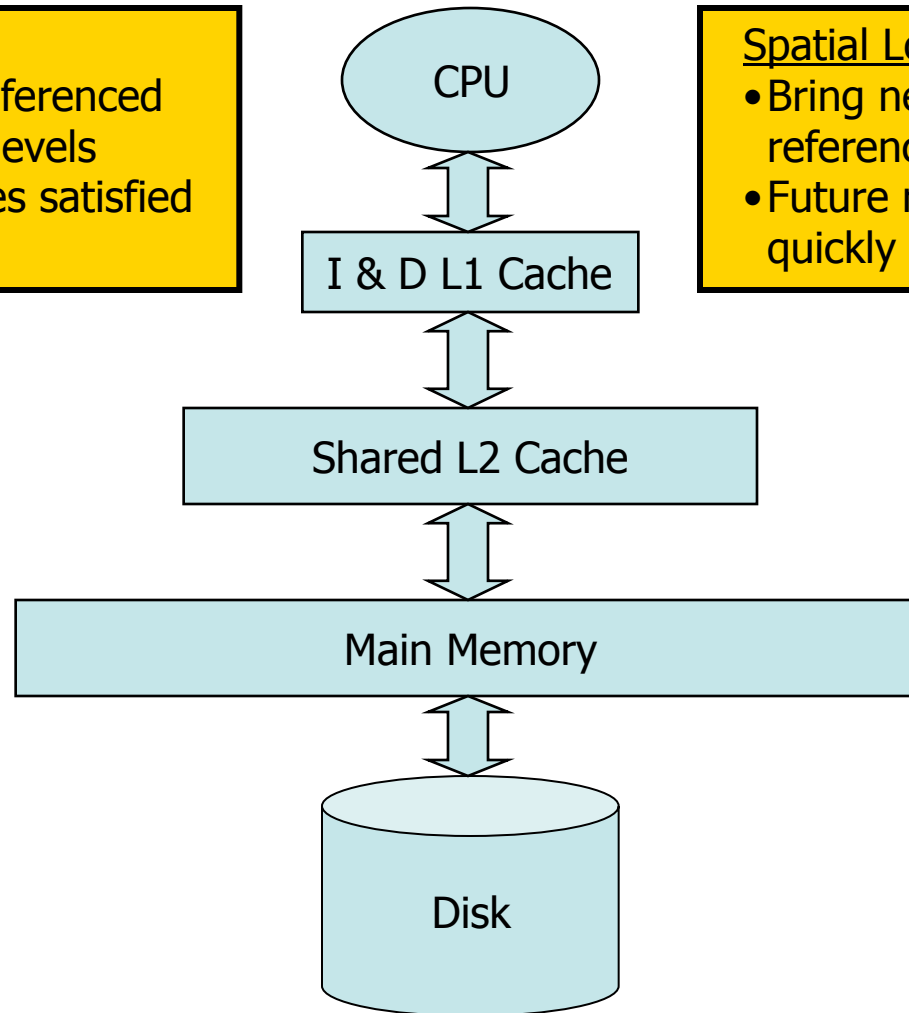
Memory Hierarchy

Temporal Locality

- Keep recently referenced items at higher levels
- Future references satisfied quickly

Spatial Locality

- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly



Performance

CPU execution time = (CPU clock cycles + memory stall cycles) x Clock Cycle time

Memory Stall cycles = Number of misses x miss penalty

= IC x misses/Instruction x miss penalty

= IC x memory access/instruction x miss rate x miss penalty



Memory Hierarchy Basics

- Four Basic Questions
 - Where can a block be placed in the upper level?
 - Block Placement
 - How a block found if it is in the upper level?
 - Block Identification
 - Which block should be replaced on miss
 - Block Replacement
 - What happens on write
 - Write Strategy



Four Burning Questions

- These are:
 - Placement
 - Where can a block of memory go?
 - Identification
 - How do I find a block of memory?
 - Replacement
 - How do I make space for new blocks?
 - Write Policy
 - How do I propagate changes?
- Consider these for caches
 - Usually SRAM
- Will consider main memory, disks later



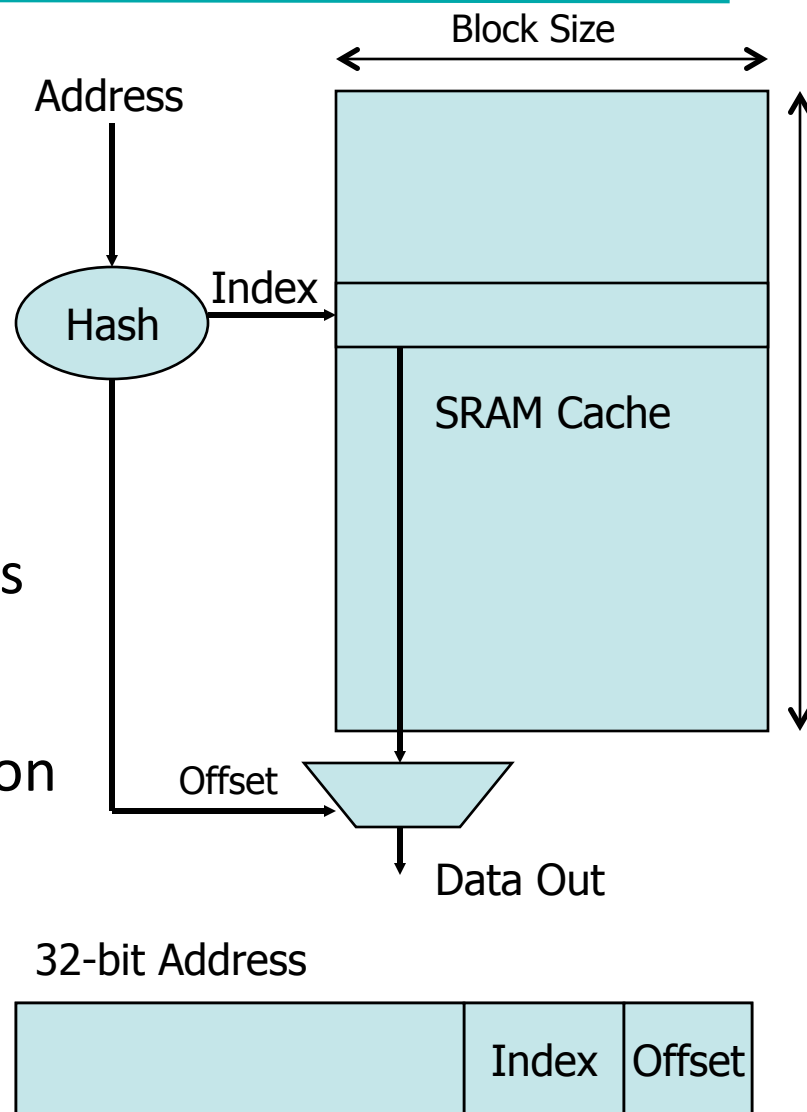
Placement

Memory Type	Placement	Comments
Registers	Anywhere; Int, FP, SPR	Compiler/programmer manages
Cache (SRAM)	Fixed in H/W	<i>Direct-mapped, set-associative, fully-associative</i>
DRAM	Anywhere	O/S manages
Disk	Anywhere	O/S manages



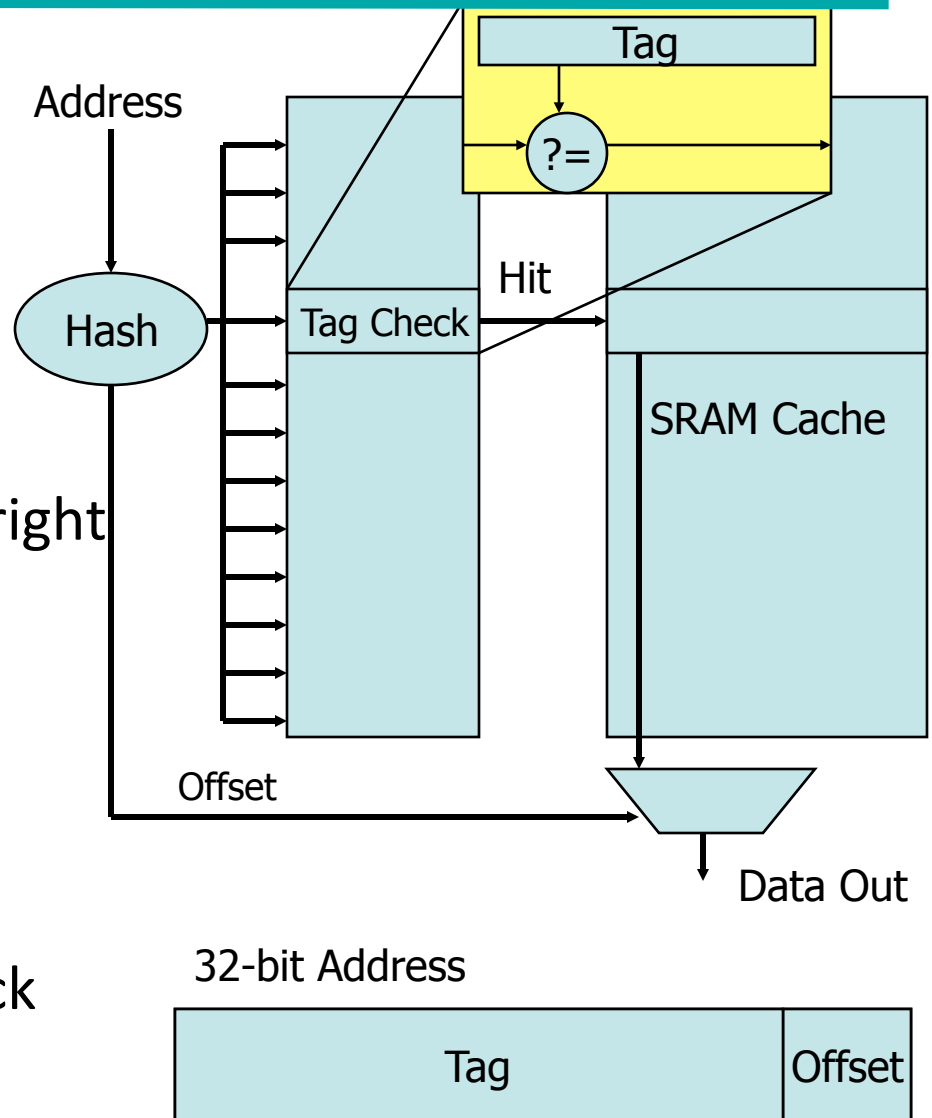
Placement

- Address Range
 - Exceeds cache capacity
- Map address to finite capacity
 - Called a *hash*
 - Usually just masks high-order bits
- *Direct-mapped*
 - Block can only exist in one location
 - Hash collisions cause problems



Placement

- *Fully-associative*
 - Block can exist anywhere
 - No more hash collisions
- *Identification*
 - How do I know I have the right block?
 - Called a *tag check*
 - Must store address tags
 - Compare against address
- **Expensive!**
 - Tag & comparator per block



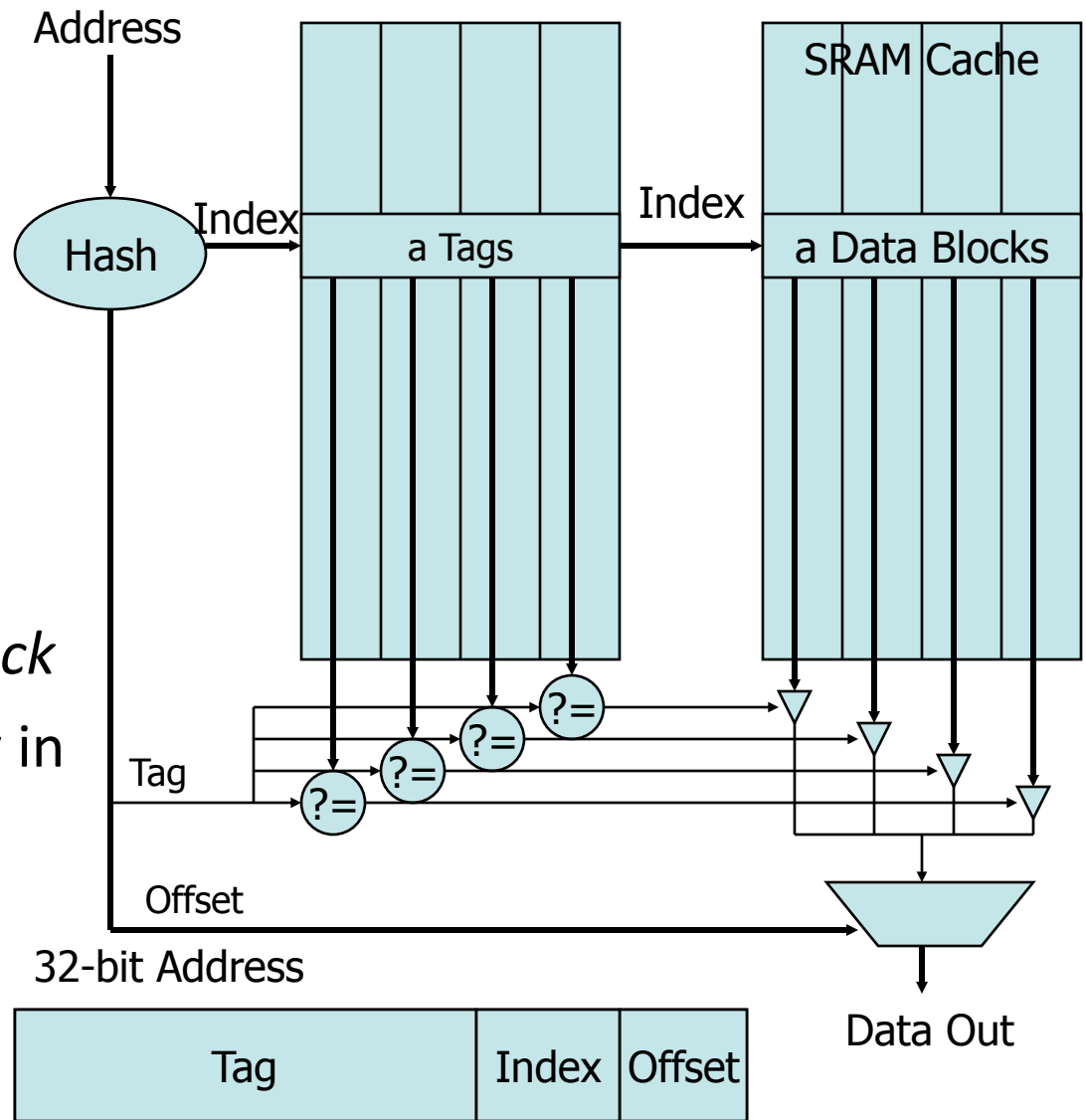
Placement

- *Set-associative*

- Block can be in a locations
- Hash collisions:
 - a still OK

- *Identification*

- Still perform *tag check*
- However, only a few in parallel



Placement and Identification

32-bit Address



Portion	Length	Purpose
Offset	$o = \log_2(\text{block size})$	Select word within block
Index	$i = \log_2(\text{number of sets})$	Select set of blocks
Tag	$t = 32 - o - i$	ID block within set

- Consider: $\langle BS = \text{block size}, S = \text{sets}, B = \text{blocks} \rangle$
 - $\langle 64, 64, 64 \rangle$: $o = 6, i = 6, t = 20$: direct-mapped ($S = B$)
 - $\langle 64, 16, 64 \rangle$: $o = 6, i = 4, t = 22$: 4-way S-A ($S = B / 4$)
 - $\langle 64, 1, 64 \rangle$: $o = 6, i = 0, t = 26$: fully associative ($S = 1$)
- Total size = $BS \times B = BS \times S \times (B/S)$



Replacement

- Cache has finite size
 - What do we do when it is full?
- Analogy: desktop full?
 - Move books to bookshelf to make room
- Same idea:
 - Move blocks to next level of cache



Replacement

- How do we choose *victim*?
 - Verbs: *Victimize, evict, replace, cast out*
- Several policies are possible
 - FIFO (first-in-first-out)
 - LRU (least recently used)
 - NMRU (not most recently used)
 - Pseudo-random (yes, really!)
- Pick victim within *set* where $a = \text{associativity}$
 - If $a \leq 2$, LRU is cheap and easy (1 bit)
 - If $a > 2$, it gets harder
 - Pseudo-random works pretty well for caches



Write Policy

- Memory hierarchy
 - 2 or more copies of same block
 - Main memory and/or disk
 - Caches
- What to do on a write?
 - Eventually, all copies must be changed
 - Write must *propagate* to all levels



Write Policy

- Easiest policy: *write-through*
- Every write propagates directly through hierarchy
 - Write in L1, L2, memory, disk (?!?)
- Why is this a bad idea?
 - Very high bandwidth requirement
 - Remember, large memories are slow
- Popular in real systems only to the L2
 - Every write updates L1 and L2
 - Beyond L2, use *write-back* policy



Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
 - Invalid – not present in the cache
 - Clean – present, but not written (unmodified)
 - Dirty – present and written (modified)
- Store state in tag array, next to address tag
 - Mark dirty bit on a write
- On eviction, check dirty bit
 - If set, write back dirty line to next level
 - Called a *writeback* or *castout*



Write Policy

- Complications of write-back policy
 - Stale copies lower in the hierarchy
 - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
 - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
 - Called coherent I/O
 - Must check caches for dirty copies before reading main memory



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
		0
		0



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
11		1



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss

Tag Array

Tag0	Tag1	LRU
10		1
		0
10		1
11		1



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss

Tag Array

Tag0	Tag1	LRU
10	11	0
		0
10		1
11		1



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10		1
11		1



Cache Example

- 32B Cache: $\langle BS=4, S=4, B=8 \rangle$
 - $o=2, i=2, t=2$; 2-way set-associative
 - Initially empty
 - Only tag array shown on right
- Trace execution of:

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10 d		1
11		1

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict
Store 0x29	101001	2/0	Hit/Dirty



Thank You

