CSE 5095 & ECE 4451 & ECE 5451 – Spring 2017

Lecture 5a

Caching Review

Marten van Dijk
Syed Kamran Haider, Chenglu Jin, Phuong Ha Nguyen

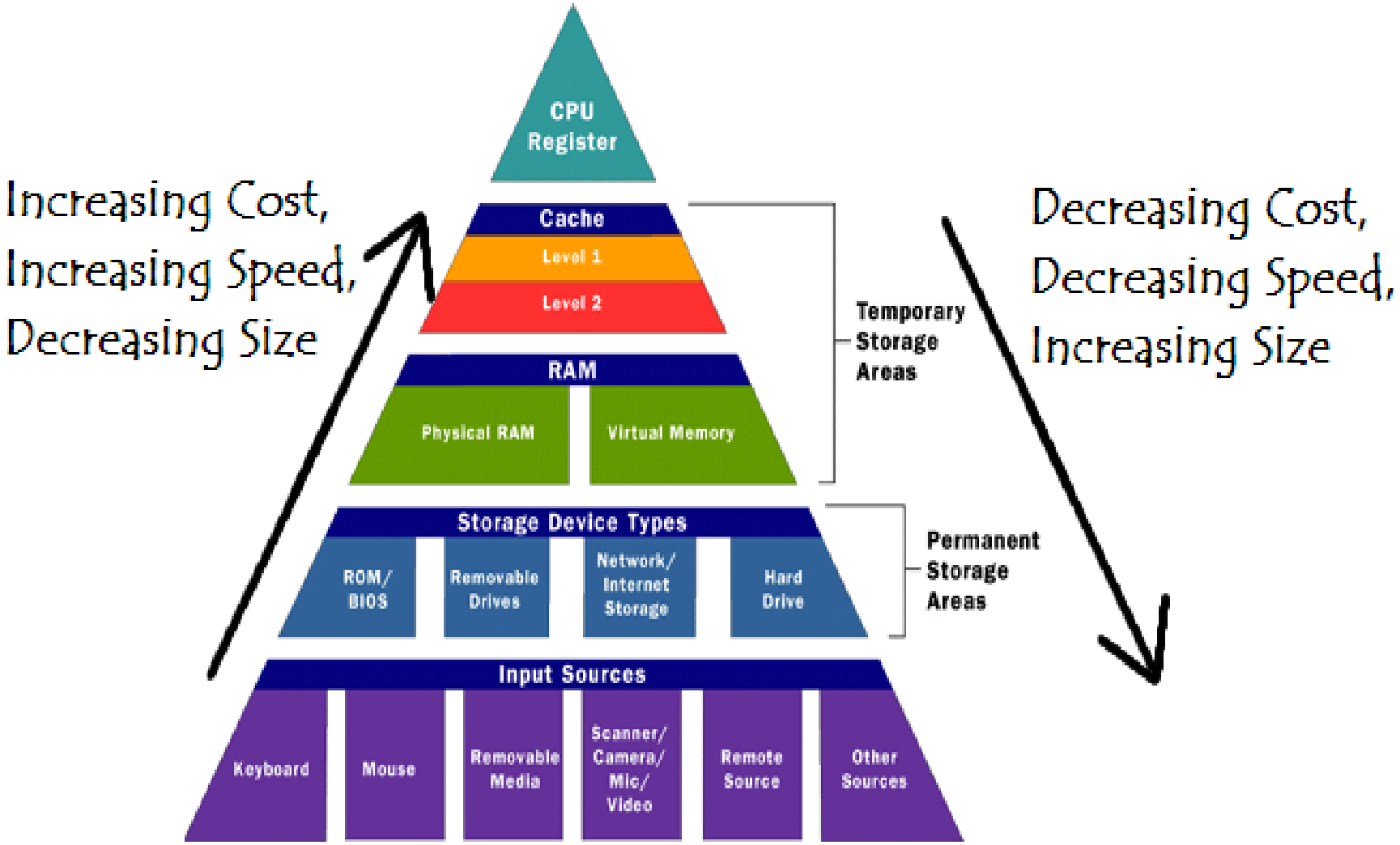
Department of Electrical & Computer Engineering
University of Connecticut

Material taken from:

1. "Cache and Virtual Memory Replacement Algorithms", slides by Michael Smaili, Spring 2008
2. "Cache Rep;lacement Policies", Mikko M. Lipasti, Spring 2012
3. "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," slides by G. E. Suh, C. W. O'Donnell, I. Sachdev, and S, Devadas

UCONN

Memory Layout



Memory Hierarchy

- Provide memories of various speed and size at different points in the system.
- Use a memory management scheme which will move data between levels.
  - Those items most often used should be stored in faster levels.
  - Those items seldom used should be stored in lower levels.

- Cache: a small, fast "buffer" that lies between the CPU and the Main Memory which holds the most recently accessed data.
- Virtual Memory: Program and data are assigned addresses independent of the amount of physical main memory storage actually available and the location from which the program will actually be executed.
- Hit ratio: Probability that next memory access is found in the cache.
- Miss rate: (1.0 – Hit rate)

- Primary goal of Cache: increase Speed.
- Primary goal of Virtual Memory: increase Space.

Cache Design

- Key issues 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
- Also apply to main memory, disks

Placement and Identification

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

Fully Associative Mapping

A main memory block can map into any block in cache.

Main Memory

Block 1	000	Prog A
Block 2	001	Prog B
Block 3	010	Prog C
Block 4	011	Prog D
Block 5	100	Data A
Block 6	101	Data B
Block 7	110	Data C
Block 8	111	Data D

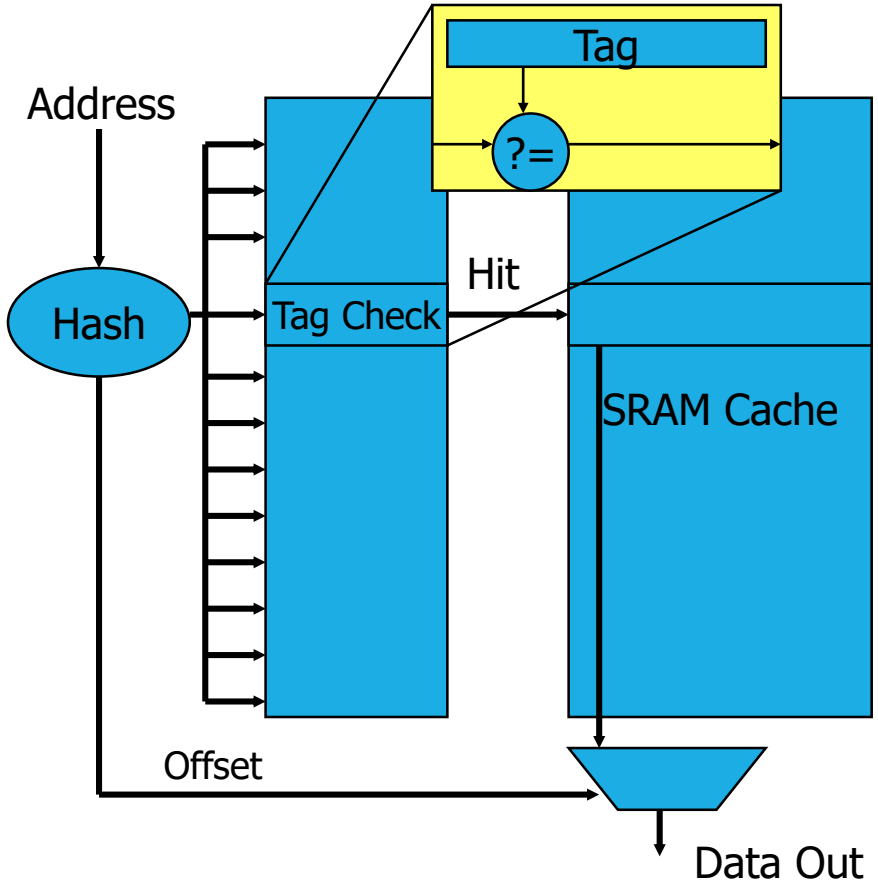
Cache Memory

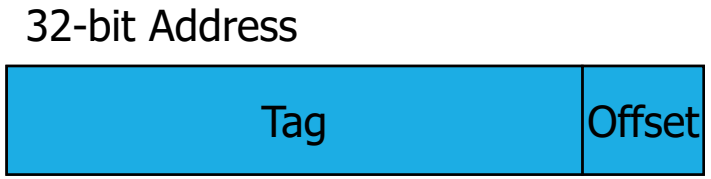
Block 1	*100*	*Data A*
Block 2	*010*	*Prog C*

Italics: Stored in Memory

Pros/Cons

- Advantages:
  - No Contention
  - Easy to implement
- Disadvantages:
  - Very expensive
  - Very wasteful of cache storage since you must store full primary memory address





Direct Mapping

Store higher order tag bits along with data in cache.

Main Memory

Block 1	000	Prog A
Block 2	001	Prog B
Block 3	010	Prog C
Block 4	011	Prog D
Block 5	100	Data A
Block 6	101	Data B
Block 7	110	Data C
Block 8	111	Data D

Cache Memory

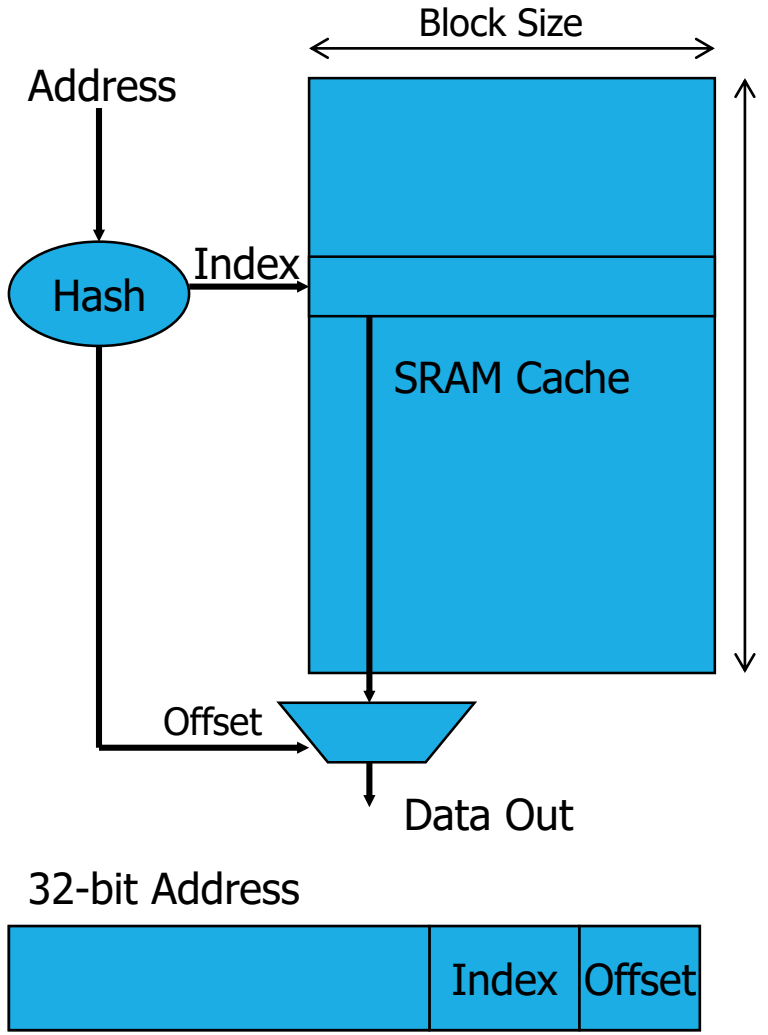
Block 1	00	*0*	*Prog A*
Block 2	01		
Block 3	10	*1*	*Data C*
Block 4	11	*0*	*Prog D*

Italics: Stored in Memory

Index bits
Tag bits

Pros/Cons

- Advantages:
  - Low cost; doesn't require an associative memory in hardware
  - Uses less cache space
- Disadvantages:
  - Contention with main memory data with same index bits.



Set Associative Mapping

Puts a fully associative cache within a direct-mapped cache.

Main Memory

Block 1	000	Prog A
Block 2	001	Prog B
Block 3	010	Prog C
Block 4	011	Prog D
Block 5	100	Data A
Block 6	101	Data B
Block 7	110	Data C
Block 8	111	Data D

Cache Memory

Set 1	0	*00*	Prog A	*10*	Data A
Set 2	1	*11*	Data D	*10*	Data B

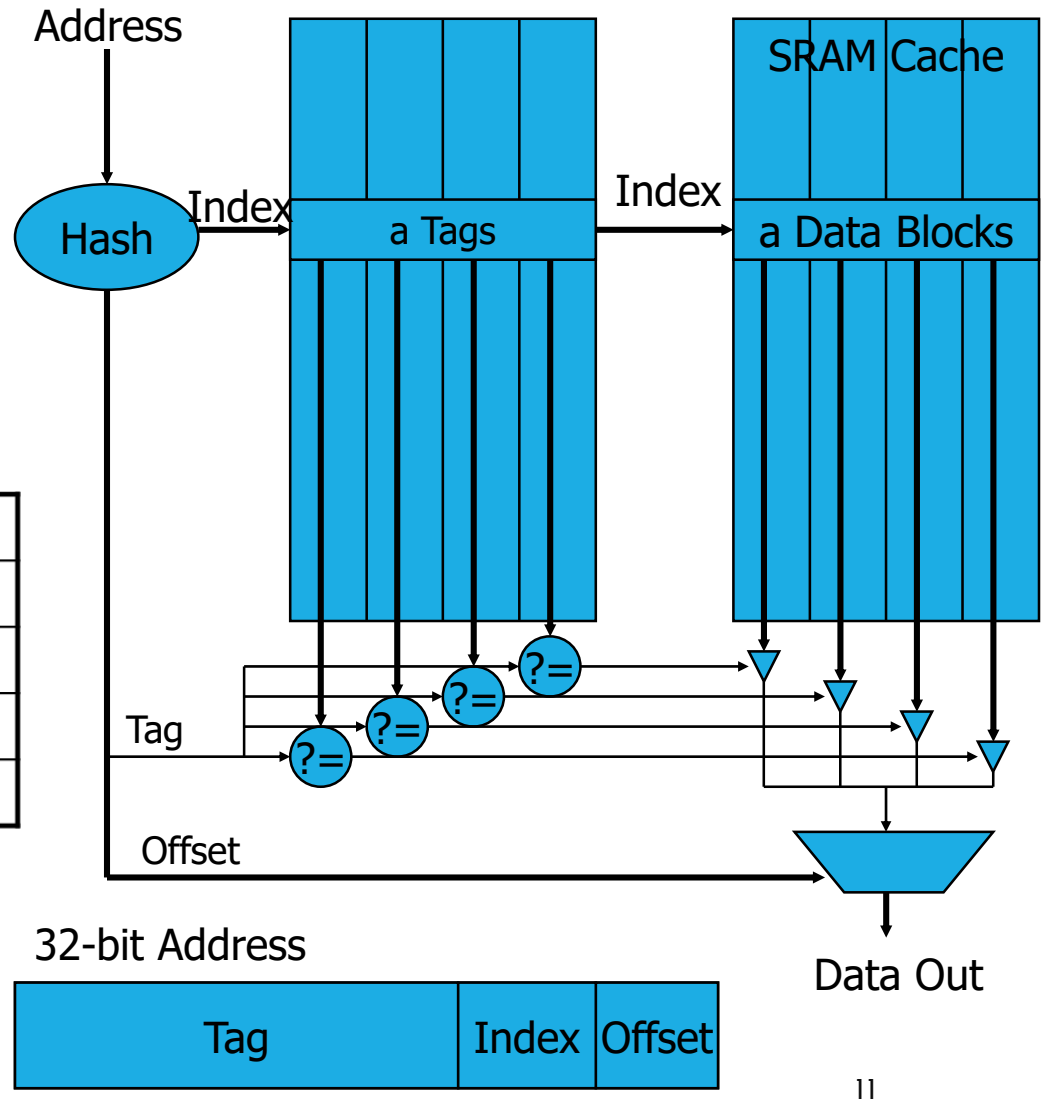
Italics: Stored in Memory

Index bits
Tag bits

Pros/Cons

- Intermediate compromise solution between Fully Associative and Direct Mapping
  - Not as expensive and complex as a fully associative approach.
  - Not as much contention as in a direct mapping approach.

Cost	Degree Associativity	Miss Rate	Delta
$	1-way	6.6%	
$$	2-way	5.4%	1.2
$$$$	4-way	4.9%	.5
$$$$$$$$	8-way	4.8%	.1



Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Many policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - NMRU (not most recently used)
  - NRU
  - Pseudo-random (yes, really!)
  - Optimal
  - Etc

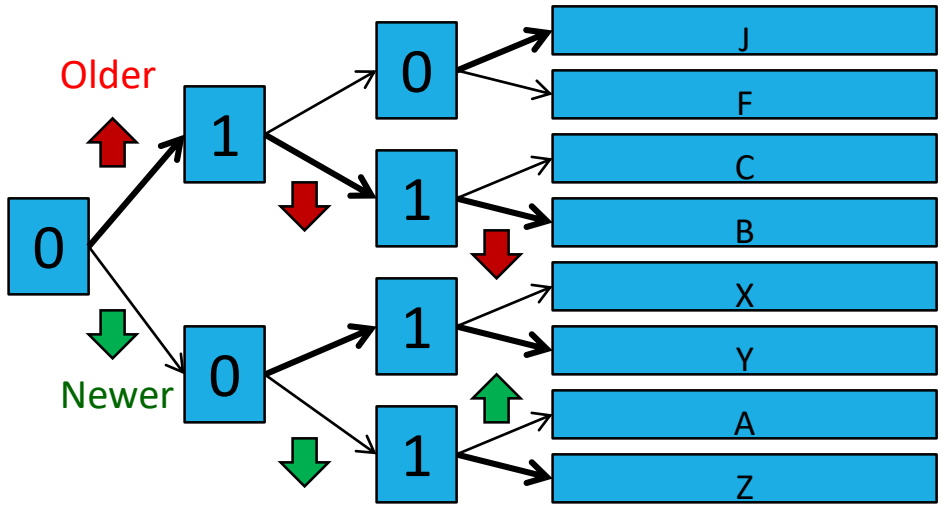
LRU

- For a=2, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access
- For a>2, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
- List overhead: $log_2(a)$ bits /block
  - Shift register implementation

LRU Implementation

- 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

Practical Pseudo LRU



- Rather than true LRU, use binary tree
- Each node records which half is older/newer
- Update nodes on each reference
- Follow older pointers to find LRU victim

True LRU Shortcomings

- Streaming data/scans: x_0, x_1, …, x_n
  - Effectively no temporal reuse
- Thrashing: *reuse distance > a*
  - Temporal reuse exists but LRU fails
- All blocks march from MRU to LRU
  - Other conflicting blocks are pushed out
- For *n>a* no blocks remain after scan/thrash
  - Incur many conflict misses after scan ends
- Pseudo-LRU sometimes helps a little bit

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
- Do we update memory on writes?
  - Write-through
    - Memory immediately updated on each write
  - Write-back
    - Memory updated when line replaced

Write-Back Caches

- Need a ***Dirty*** bit for each line
  - 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)

Prefetching

- Predict future misses and get data into cache
  - If access does happen, we have a hit now
    (or a partial miss, if data is on the way)
  - If access does not happen, ***cache pollution***
    (replaced other data with junk we don't need)
- To avoid pollution, prefetch buffers
  - Pollution a big problem for small caches
  - Have a small separate buffer for prefetches
    - When we do access it, put data in cache
    - If we don't access it, cache not polluted

Advanced Topics (not covered)

- Inclusive and exclusive caches (in a cache hierarchy)
- Blocking and non-blocking caches
- Etc.