ECE/CS 552: Cache Memory

Instructor: Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on set created by Mark Hill
Updated by Mikko Lipasti

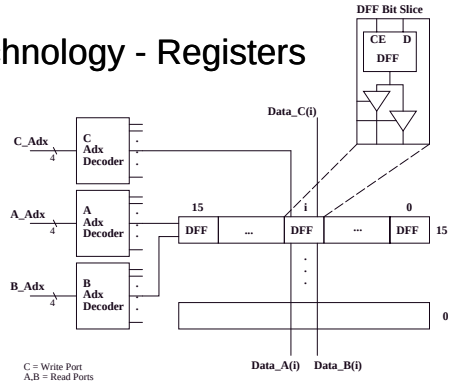
Big Picture

- Memory
  - Just an "ocean of bits"
  - Many technologies are available
- Key issues
  - Technology (how bits are stored)
  - Placement (where bits are stored)
  - Identification (finding the right bits)
  - Replacement (finding space for new bits)
  - Write policy (propagating changes to bits)
- Must answer these regardless of memory type

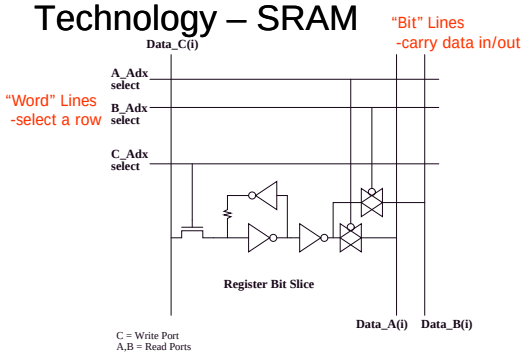
Types of Memory

Type	Size	Speed	Cost/bit
Register	< 1KB	< 1ns	$$$$
On-chip SRAM	8KB-6MB	< 10ns	$$$
Off-chip SRAM	1Mb – 16Mb	< 20ns	$$
DRAM	64MB – 1TB	< 100ns	$
Disk	40GB – 1PB	< 20ms	~0

Technology - Registers

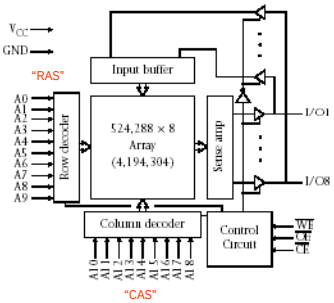


Technology – SRAM



Technology – Off-chip SRAM

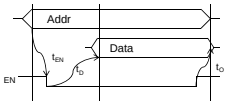
- Commodity
  - 1Mb – 16Mb
- 554 Lab SRAM
  - AS7C4096
  - 512K x 8b
    - 512KB
    - Or 4Mb
  - 10-20ns
- Note: sense amp
  - Analog circuit
  - High-gain amplifier

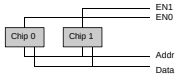


Technology – DRAM

- Logically similar to SRAM
- Commodity DRAM chips
  - E.g. 1Gb
  - Standardized address/data/control interfaces
- Very dense
  - 1T per cell (bit)
  - Data stored in capacitor – decays over time
    - Must rewrite on read, refresh
- Density improving vastly over time
- Latency barely improving

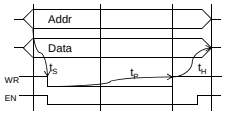
Memory Timing – Read

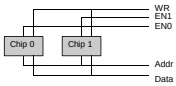




- Latch-based SRAM or asynchronous DRAM (FPM/EDO)
  - Multiple chips/banks share address bus and tristate data bus
  - Enables are decoded from address to select bank
    - E.g. bbbbbbb0 is bank 0, bbbbbbb1 is bank 1
- Timing constraints: straightforward
  - t_{EN} setup time from Addr stable to EN active (often zero)
  - t_D delay from EN to valid data (10ns typical for SRAM)
  - t_O delay from EN disable to data tristate off (nonzero)

Memory Timing - Write



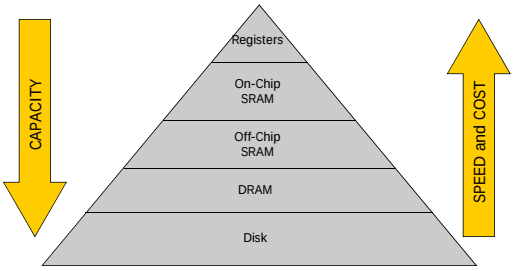


- WR & EN triggers write of Data to ADDR
- Timing constraints: not so easy
  - t_S setup time from Data & Addr stable to WR pulse
  - t_P minimum write pulse duration
  - t_H hold time for data/addr beyond write pulse end
- Challenge: WR pulse must start late, end early
  - $>t_S$ after Addr/Data, $>t_H$ before end of cycle
  - Requires multicycle control or glitch-free clock divider

Technology – Disk

- Covered in more detail later (input/output)
- Bits stored as magnetic charge
- Still mechanical!
  - Disk rotates (3600-15000 RPM)
  - Head seeks to track, waits for sector to rotate to it
  - Solid-state replacements in the works
    - MRAM, etc.
- Glacially slow compared to DRAM (10-20ms)
- Density improvements astounding (100%/year)

Memory Hierarchy



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

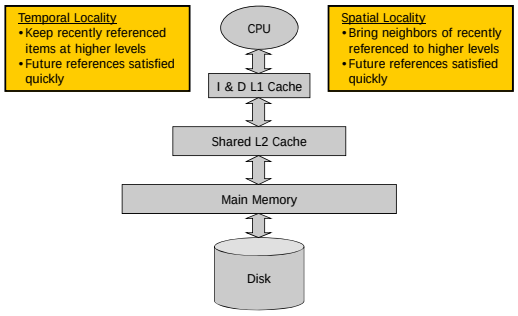
Why Does a Hierarchy Work?

- 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



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

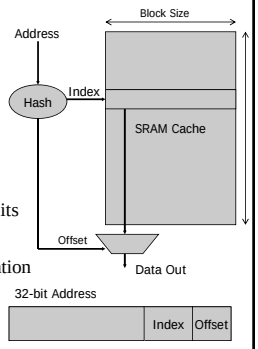
Placement

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

HUH?

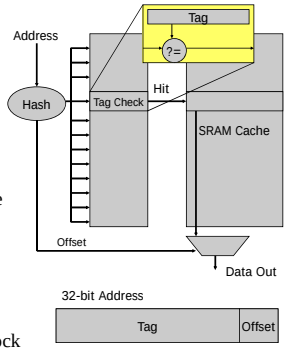
Placement

- 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



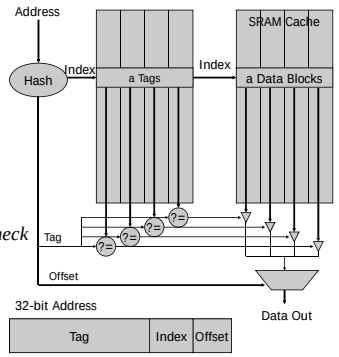
Placement

- *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 in parallel



Placement and Identification

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

- Consider: <BS=block size, S=sets, B=blocks>
  - <64,64,64>: o=6, i=6, t=20: direct-mapped (S=B)
  - <64,16,64>: o=6, i=4, t=22: 4-way S-A (S = B / 4)
  - <64,1,64>: o=6, i=0, t=26: fully associative (S=1)
- Total size = BS x B = BS x S x (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 = *associativity*
  - If a <= 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - 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: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

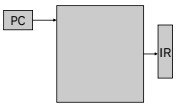
Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict
Store 0x29	101001	2/0	Hit/Dirty

Tag Array

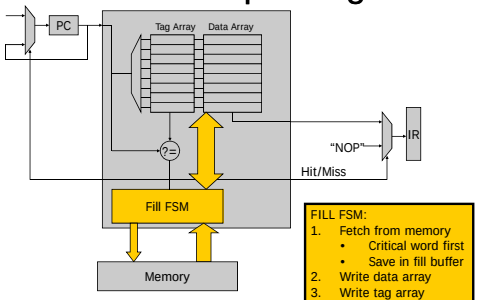
Tag0	Tag1	LRU
01	11	1
		0
10 d		1
11		1

Caches and Pipelining

- Instruction cache
  - No writes, so simpler
- Interface to pipeline:
  - Fetch address (from PC)
  - Supply instruction (to IR)
- What happens on a miss?
  - Stall pipeline; inject nop
  - Initiate cache fill from memory
  - Supply requested instruction, end stall condition



I-Caches and Pipelining



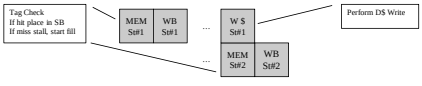
D-Caches and Pipelining

- Pipelining loads from cache
  - Hit/Miss signal from cache
  - Stalls pipeline or inject NOPs?
    - Hard to do in current real designs, since wires are too slow for global stall signals
  - Instead, treat more like branch misprediction
    - Cancel/flush pipeline
    - Restart when cache fill logic is done

D-Caches and Pipelining

- Stores more difficult
  - MEM stage:
    - Perform tag check
    - Only enable write on a hit
    - On a miss, must not write (data corruption)
  - Problem:
    - Must do tag check and data array access sequentially
    - This will hurt cycle time or force extra pipeline stage
    - Extra pipeline stage delays loads as well: IPC hit!

Solution: Pipelining Writes



- Store #1 performs tag check only in MEM stage
  - <value, address, cache way> placed in store buffer (SB)
- When store #2 reaches MEM stage
  - Store #1 writes to data cache
- In the meantime, must handle RAW to store buffer
  - Pending write in SB to address A
  - Newer loads must check SB for conflict
  - Stall/flush SB, or forward directly from SB
- Any load miss must also flush SB first
  - Otherwise SB D$ write may be to wrong line
- Can expand to >1 entry to overlap store misses

Summary

- Memory technology
- Memory hierarchy
  - Temporal and spatial locality
- Caches
  - Placement
  - Identification
  - Replacement
  - Write Policy
- Pipeline integration of caches