Computer Architecture (Memory Systems: RAM, Cache & Performance Measures)

विद्याधनं सर्वधन प्रधानम्

Subhasis Bhattacharjee
Department of Computer Science and Engineering,
Indian Institute of Technology, Jammu

September 18, 2023

Outline

Overview of the Memory System

Need for a Fast Memory System

- We have up till now assumed that the memory is one large array of bytes
  - Starts a 0, and ends at (232 – 1)
  - Takes 1 cycle to access memory (read/write)
- All programs share the memory
  - We somehow magically avoid overlaps between programs running on the same processor
  - All our programs require less than 4 GB of space

Regarding all the memory being homogeneous $\rightarrow$ NOT TRUE

Typical Characteristics of Different Memory Devices		
Cell Type	Area	Typical Latency
Master Slave D flip flop	0.8 μm^2	Fraction of a cycle
SRAM cell in an array	0.08 μm^2	1-5 cycles
DRAM cell in an array	0.005 μm^2	50-200 cycles

Should we make our memory using only flip-flops?

- 10X the area of a memory with SRAM cells
- 160X the area of a memory with DRAM cells
- Significantly more power.

Tradeoffs

Tradeoffs

- Area, Power, and Latency
- Increase Area $\rightarrow$ Reduce latency, increase power
- Reduce latency $\rightarrow$ increase area, increase power
- Reduce power $\rightarrow$ reduce area, increase latency
- We cannot have *the best of all worlds*

What do we do ?

- We cannot create a memory of just flip flops
  - We will hardly be able to store anything
- We cannot create a memory of just SRAM cells
  - We need more storage, and we will not have a 1 cycle latency
- We cannot create a memory of DRAM cells
  - We cannot afford 50+ cycles per access7

Memory Access Latency

What does memory access latency depend on ?

- Size of the memory $\rightarrow$ larger is the size, slower it is
- Number of ports $\rightarrow$ More are the ports (parallel accesses/cycle), slower is the memory
- Technology used $\rightarrow$ SRAM, DRAM, flip-flops

Temporal and Spatial Locality

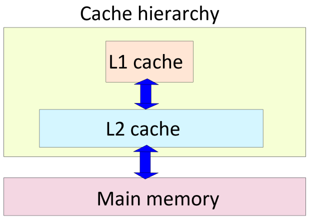
Spatial Locality

It is a concept that states that if a resource is accessed at some point of time, then most likely similar resources will be accessed again in the near future.

Temporal Locality

It is a concept that states that if a resource is accessed at some point of time, then most likely it will be accessed again in a short period of time.

Exploiting Temporal Locality



- Use a hierarchical memory system
- L1 (SRAM cells), L2 (SRAM cells), Main Memory (DRAM cells)

Storage to Random Access Memory

- Sequential circuits depend upon the presence of memory.
- A flip-flop can store one bit of information.
- A register can store a single “word,” typically 32-64 bits.
- RAM allows us to store even larger amounts of data.
- Temporary storage in a computer system.

Required Capabilities of Storage

- We should be able to store a value.
- We should be able to read the value that was saved.
- We should be able to change the stored value.

Operation on a RAM

Doing Operation on a RAM

- An address will specify which memory value we're interested in.
- Each value can be a multiple-bit word (e.g., 32 bits).
- We'll refine the memory properties as follows:

- Store many words, one per address
- Read the word that was saved at a particular address
- Change the word that's saved at a particular address

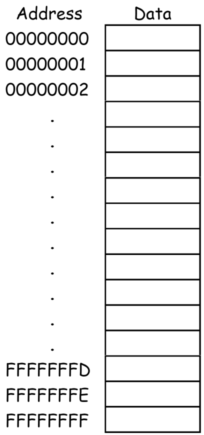
Two types of RAM

Two types of RAM

- SRAM $\rightarrow$ Static Random Access Memory
- DRAM $\rightarrow$ Dynamic Random Access Memory

Visualization of RAM - Human Perception

- You can think of computer memory as being one big array of data.
- The address serves as an array index.
- Each address refers to one word of data.
- You can read or modify the data at any given memory address, just like you can read or modify the contents of an array at any given index.



Back To Memory System Organization

The Caches

- The **L1 cache** is a small memory (8-64 KB) composed of SRAM cells
- The **L2 cache** is larger and slower (128 KB
- 4 MB) (SRAM cells)
- The **main memory** is even larger (1 – 64 GB) (DRAM cells)

Cache hierarchy:

- The main memory contains all the memory locations
- The caches contain a subset of memory locations

Inclusive Cache Hierarchy:

addresses(L1) $\subset$ addresses(L2) $\subset$ addresses(main memory)

Access Protocol

Access Protocol

- First access the L1 cache. If the memory location is present, we have a cache hit. (Goal: maximizing cache hit)
  - Perform the access (read/write)
- Otherwise, we have a cache miss. (Goal: minimizing cache miss)
  - Fetch the value from the lower levels of the memory system, and populate the cache.
  - Follow this protocol recursively

Advantage

Typical Hit Rates, Latencies

- L1 : 95%, 1 cycle
- L2 : 60%, 10 cycles
- Main Memory : 100%, 300 cycles

Answer:

- 95% of the memory accesses take a single cycle
- 3% take, 10 cycles
- 2% take, 300 cycles

Exploiting Spatial Locality

Observation from the address locality plot

Most of the addresses are within +/- 25 bytes.

Idea for Improving Memory Performance

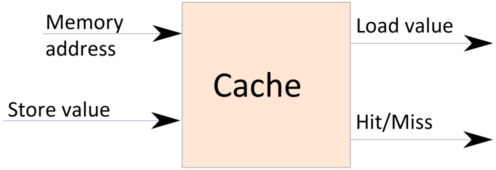
- Group memory addresses into sets of n bytes
- Each group is known as a cache line or cache block
- A cache block is typically 32, 64, or 128 bytes

Benefit:

Once we fetch a block of 32/64 bytes. A lot of accesses in a short time interval will find their data in the block.

Caches

Overview of a Basic Cache



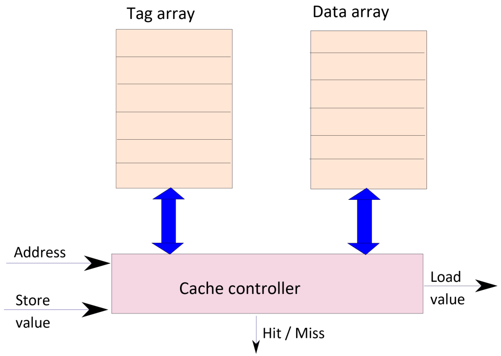
Cache Idea:

- Saves a subset of memory values
- We can either have hit or miss
- The load/store is successful if we

Basic Cache Operations

- **lookup** $\rightarrow$ Check if the memory location is present
- **data read** $\rightarrow$ read data from the cache
- **data write** $\rightarrow$ write data to the cache
- **insert** $\rightarrow$ insert a block into a cache
- **replace** $\rightarrow$ find a candidate for replacement
- **evict** $\rightarrow$ throw a block out of the cache

Cache Lookup - How do we do it?



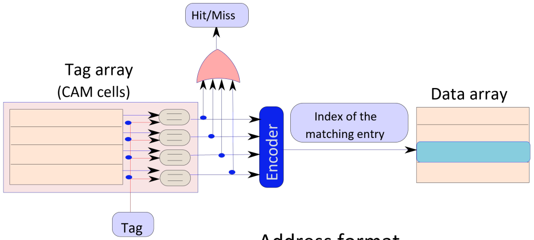
Simple Cache Structure

- Let us have **two SRAM arrays**
- **tag array** $\rightarrow$ Saves a part of the block address such that the block can be uniquely identified
- **block array** $\rightarrow$ Saves the contents of the block
- Both the arrays have the same number of entries

Fully Associative Cache

Example : 8 KB Cache, block size of 64 bytes, 32 bit memory system

- We have 213 / 26 = 128 entries
- A block can be saved in any entry
- 26 bit tag, and 6 bit offset



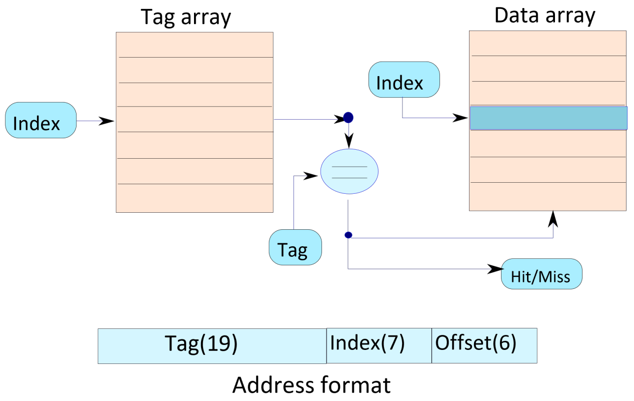
Address format

Tag	Offset

Fully Associative Cache - Implementation

- We use an array of CAM cells for the tag array
- Each entry compares its contents with the tag
- Sets the match line to 1
- The OR gate computes a hit or miss
- The encoder computes the index of the matching entry.
- We then read the contents of the matching entry from the block array

Direct Mapped Cache

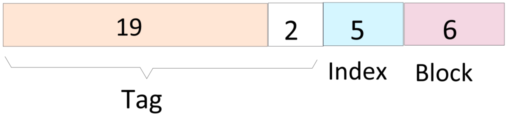


Each block can be mapped to only 1 entry

Direct Mapped Cache

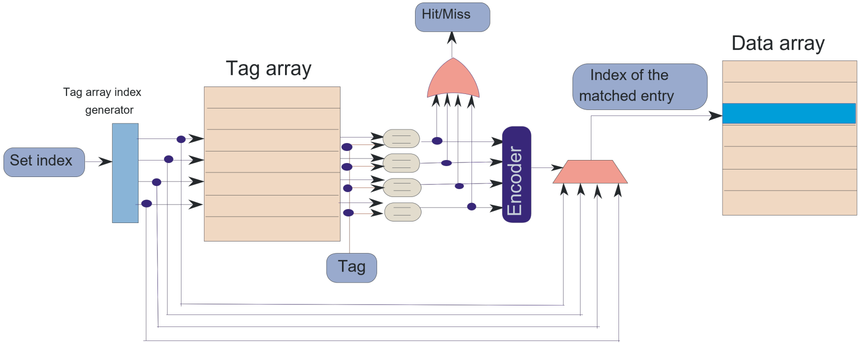
- We have 128 entries in our cache.
- We compute the index as idx = block address % 128
- We access entry, idx, in the tag array and compare the contents of the tag (19 msb bits of the address)
- If there is a match $\rightarrow$ hit
- else $\rightarrow$ miss
- Need a solution that is in the middle of the spectrum

Set Associative Cache - Tag & Block selection



- Let us assume that an address can reside in 4 locations
  - Access all 4 locations, and see if there is a hit
- Thus, we have $128/4 = 32$ indices
- Each index points to a set of 4 entries
- $\rightarrow$ We now use a 21 bit tag, 5 bit index

Set Associative Cache



Set Associative Cache - Access Mechanism

- Let the **index** be i , and the **number of elements** in a set be k
  - We access indices, i*k, i*k+1 ,.., i*k + (k-1)
  - Read all the tags in the set
  - Compare the tags with the tag obtained from the address
  - Use an OR gate to compute a hit/ miss
  - Use an encoder to find the index of the matched entry51
- Read the corresponding entry from the block array
- Each entry in a set is known as a way
- A cache with k blocks in a set is known as a **k-way associative cache**

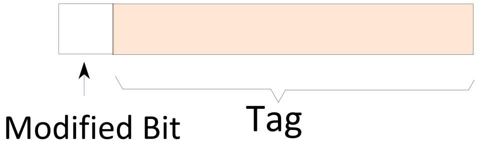
Data read operation

- This is a regular SRAM access.
- Note that the data read and lookup can be overlapped for a load access
- We can issue a parallel data read to all the ways in the cache
- Once, we compute the index of the matching tag, we can choose the correct result with a multiplexer.

Data write operation

- Before we write a value
  - We need to ensure that the block is present in the cache
- Why ?
  - Otherwise, we have to maintain the indices of the bytes that were written to
  - We treat a block as an atomic unit
  - Hence, on a miss, we fetch the entire block first
- Once a block is there in the cache
  - Go ahead and write to it ....

Modified bit



- Maintain a modified bit in the tag array.
- If a block has been written to, after it was fetched, set it to 1.

Write Policies - Two types

Write through

Whenever we write to a cache, we also write to its lower level.
Advantage: Can seamlessly evict data from the cache.

Write back

- We do not write to the lower level immediately.
- Whenever we write, we set the modified bit.
- At the time of eviction of the line, we check the value of the modified bit

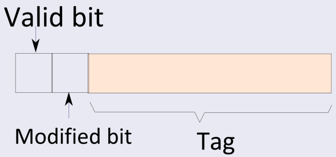
insert operation

Requirement:

If we don't find a block in a cache, We fetch it from the lower level. Then we insert the block in the cache.

Updated Tag Field

- Let us add a **valid bit** to a tag
- If the line is non-empty, valid bit is 1, Else it is 0.



insert operation - Method

- Precondition: The block is NOT in the cache.
- Find the block in the lower level
- Check if the set has an invalid line
  - If there is one, then write the fetched line to that location, set the valid bit to 1.
  - Otherwise, find a candidate for replacement (*we will see next...*)

The replace operation

Cache replacement scheme

A cache replacement scheme or replacement policy is a method to replace an entry in the set by a new entry.

Several Replacement Schemes

- Random replacement scheme
- FIFO (First In First Out) scheme
- LRU (Least Recently Used) scheme
- Psuedo-LRU scheme

FIFO - Replacement Schemes

- When we fetch a block, assign it a counter value equal to 0
- Increment the counters of the rest of the ways
- For replacement, choose the way with the highest counter (oldest).

Problems in FIFO Replacement Schemes

- Can violate the principle of temporal locality
- A line fetched early might be accessed very frequently.

LRU (least recently used)

- Replace the block that has been accessed the least in the recent past
- Most likely we will not access it in the near future
- Directly follows from the definition of stack distance
- Sadly, we need to do more work per access
- Proved to be optimal in some restrictive scenarios

Problems in LRU Replacement Schemes

- True LRU requires saving a hefty timestamp with every way
- Usually implement pseudo-LRU

Psuedo-LRU

- Let us try to mark the most recently used (MRU) elements.
- Let us associate a 3 bit counter with every way.
  - Whenever we access a line, we increment the counter.
  - We stop incrementing beyond 7.
  - We periodically decrement all the counters in a set by 1.
  - Set the counter to 7 for a newly fetched block
  - For replacement, choose the block with the smallest counter.

evict Operation

Depends on the type of cache.

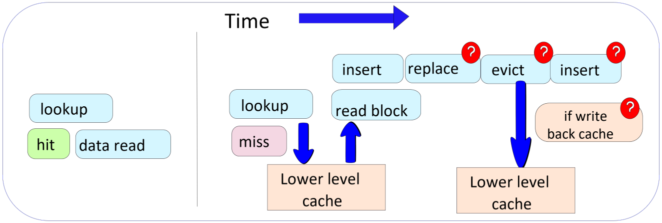
Write-through cache

Nothing needs to be done.

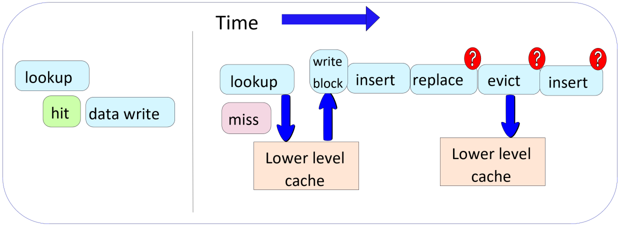
Write-back cache

- Check if the modified bit is 1
    - Write the line to the lower level
- Else, Nothing needs to be done.

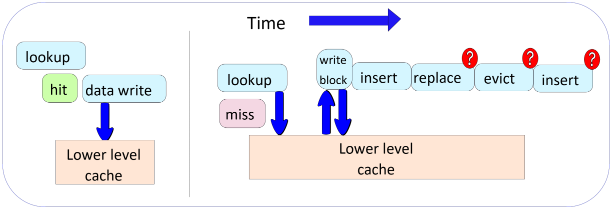
The read (load) Operation



Write operation in a write-back cache



Write operation in a write-through cache


Time
lookup
hit
data write
Lower level cache
lookup
miss
write block
insert
replace
evict
insert
Lower level cache