CMPUT 229 Lab 3

Cache Simulator

Author: Max Leontiev

Introduction

- Lab solution simulates a two-way set associative cache
  - Simulates updates to LRU bits, valid bits, tags
  - Tracks number of hits and misses, logging them as they occur
- **Input:** list of memory accesses (loads/stores with addresses)
  - Represents accesses that a program performs during its runtime
- **Output:** each hit and miss is logged, and the total number of hits, misses, and hit rate is printed at the end of the simulation
- **Goal:** understand how caches work

Background

Locality

- Programs perform many (billions!) of memory accesses during runtime, but …
- **Principle of locality:** At any *instant* in time, programs only access a small subset of available memory
  - **Temporal locality (locality in time):** If an item is referenced, it is likely to be referenced again soon
  - **Spatial locality (locality in space):** If an item is referenced, items whose addresses are close by are likely to be referenced soon

Memory Hierarchy

- Memory hierarchy uses principle of locality to provide fast access to memory (on average), while minimizing hardware cost
- **Memory Hierarchy (textbook definition):**
  A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase while the cost per bit decreases.

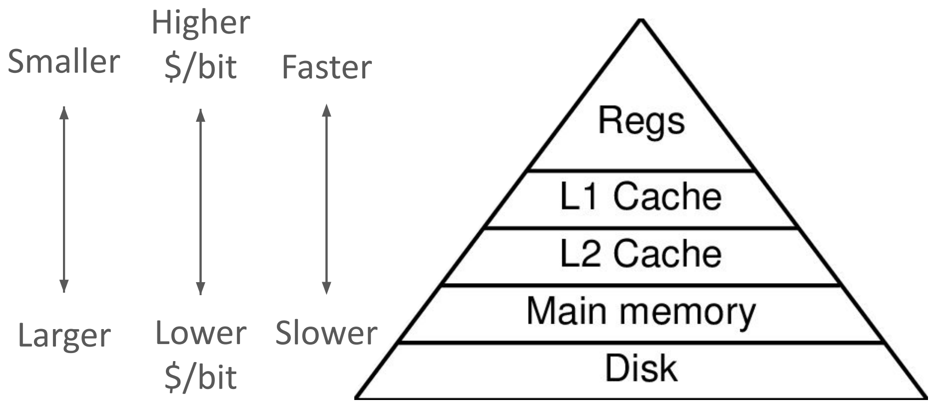


Image source: https://cgi.cse.unsw.edu.au/~reports/papers/0321.pdf

Cache Memory

- **Cache Memory:** High-speed memory that is small in size (on the order of KB or MB) and high in the memory hierarchy
  - Stores a subset of main memory data that processor is likely to need soon
  - May store data that has recently been written, depending on caching scheme
- **Block AKA line:** Unit of transfer between levels of hierarchy
  - Consists of multiple (eg. 4) words
- **Block size:** Number of words/block

Cache Hits and Misses

- **Cache hit:** When processor finds requested data in the upper level of memory hierarchy
- **Cache miss:** When processor does not find requested data in upper level of memory hierarchy
    - Requested data must be fetched from a lower level
- **Hit rate:** Fraction of memory accesses satisfied by upper level
    - *Example:* 95% L1 (level 1) cache hit rate
        - 95% of memory accesses satisfied by L1 cache
        - Requests to lower level in hierarchy are required for remaining 5% of memory accesses
- **Miss rate:** Fraction of memory accesses not satisfied by upper level
    - Miss rate is 1 - hit rate; if hit rate is 95%, miss rate is 5%

Cache Organization

Associativity

- **Associativity:** Number of locations in the cache where a given block of memory can be stored
- In an **n-way set associative** cache, each memory address maps to a **set** of blocks in the cache, and the data at that address can be in any of the blocks in the set
  - Each set contains n blocks, also known as **ways**
- This lab simulates a two-way set associative cache
  - Cache consists of many sets which each contain two blocks

Finding an address in a set associative cache

- Address of access is split up into the **tag**, **index**, and **offset**
- **Index** identifies set that requested address maps to
- **Tag** is used to look over the blocks in the set and determine if there is valid data in the cache for the requested address
  - Each block has a **valid bit** which is 1 if block contains valid data
  - Cache hit occurs if valid bit of block is 1 and tag of block matches tag of access
- **Offset** is used to identify where requested word is located within the block by acting as a byte offset from the beginning of the block
  - If word size is 4 bytes, then offset = 0 corresponds to first word in block, offset = 4 corresponds second word in block, etc.

Finding an address in a set associative cache: example

- *Example:* 32 KB 2-way set associative cache with block size of 4 on 32-bit machine
    - 18 tag bits, 10 index bits, 4 offset bits (see lab description for how to calculate this)
- Request address: **`0xB1E2E3F4`**

Tag (bits 31-14)	Index (bits 13-4)	Offset (bits 3-0)
`1011 0001 1110 0010 11`	`10 0011 1111`	`0100`

Replacement Policy

- Capacity of cache is a lot smaller than main memory
  - Data in cache must occasionally be replaced with data that is needed in the moment
  - **Replacement policy** is used to decide which data to replace
- **Least-Recently Used (LRU) replacement policy:** Replace blocks that were used (read/written to) *least* recently
  - According to principle of temporal locality, most recently used blocks are likely to be used again soon, so keep them in the cache
  - This is the policy used in this lab

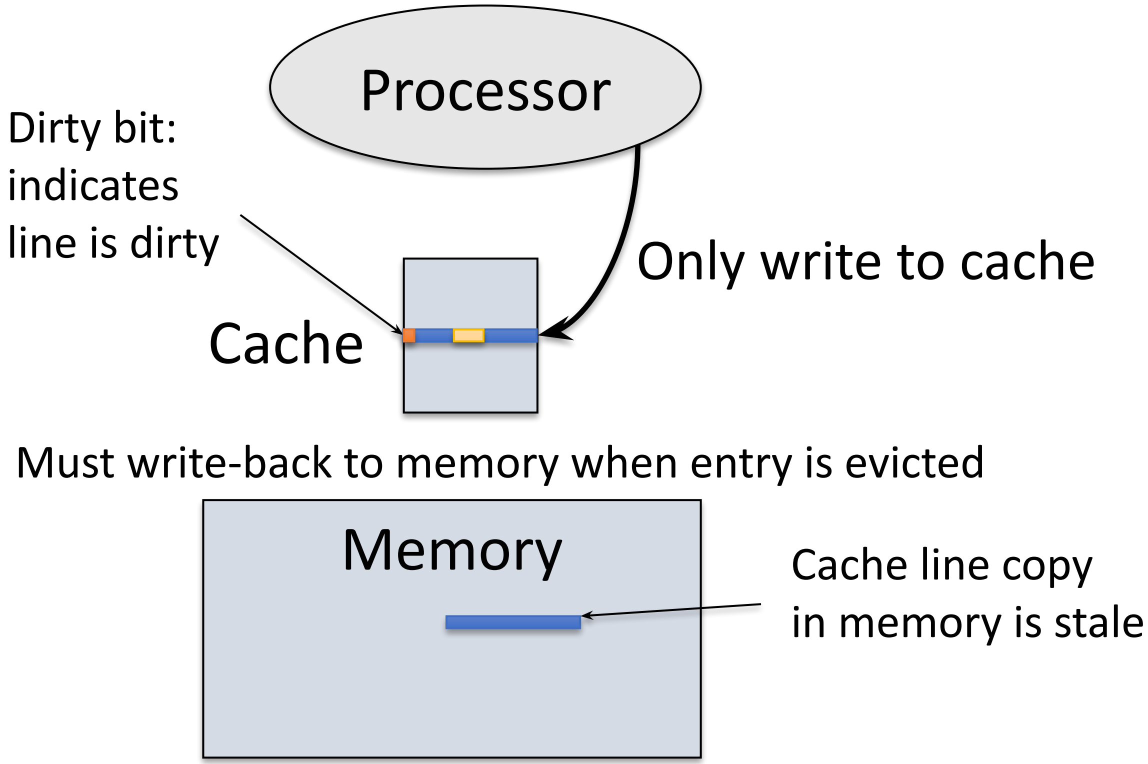
LRU Replacement Policy Implementation

- In an n-way set associative cache, implementing an LRU replacement policy requires a mechanism to keep track of which of the n ways in a set was used least recently
- When a new block is fetched into that set, the least-recently used block in the set is replaced
- In a two-way set associative cache (like in this lab), the LRU policy is implemented with a single **LRU bit** for each set that indicates which of the blocks in the set was used least recently
    - If block 0 was used least recently, LRU bit is 0
    - If block 1 was used least recently, LRU bit is 1

Write Strategy

- **Write strategy:** Policy for dealing with writes (stores)
- **Write hit:** When processor writes to a block that is in the cache
- **Write miss:** When processor writes to a block that is not in the cache
- In this lab:
  - The **write-back** policy is used on a write hit
  - The **write-allocate with write-back** policy is used on a write miss

Write Strategy (on a hit): the write-back policy

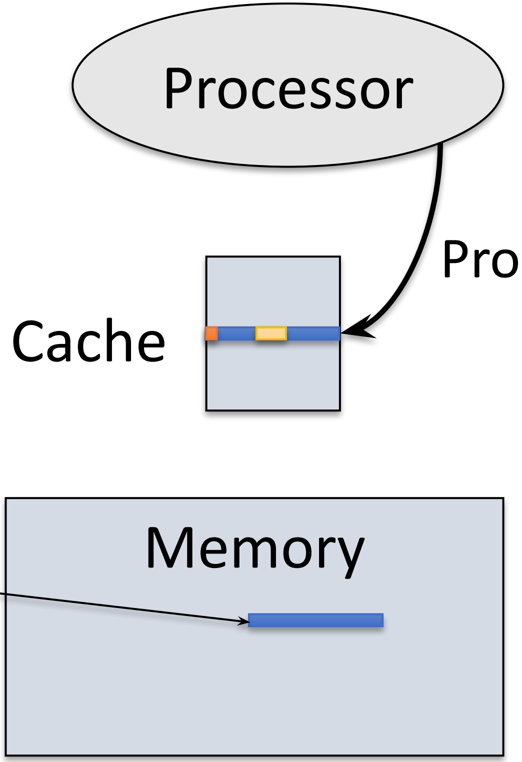


Write Strategy (on a miss): write-allocate with write-back

Now line is dirty

Processor issues write

But line is not in the cache



Cache line copy
in memory is stale

Must fetch line from memory

And then write on it

Your Task: Creating a Cache Simulator

Cache Simulator

- This lab consists of implementing **four functions** that simulate a two-way set associative cache given a list of requests (loads/stores with addresses):
  - **`simulateCache`**
  - **`simulateRequest`**
  - **`simulateHit`**
  - **`simulateMiss`**
- These functions use the **`requests`**, **`tags`**, and **`meta_bytes`** arrays provided by **`common.s`**

simulateCache

```
# ----------------------------------------------------------------------------
# simulateCache:
#
# Simulates a two-way set associative cache.
#
# simulateCache must loop over the requests array, calling simulateRequest
# for each request and counting the number of hits and misses that occur.
# simulateCache must return the total number of hits and the total number of
# misses.
#
# Args:
#   a0: N, the number of requests to memory in the input
#
# Returns:
#   a0: the total number of hits that occurred
#   a1: the total number of misses that occurred
#
# Register usage:
#   insert your register usage for this function here
# ----------------------------------------------------------------------------
```

- Loops over **`requests`** array
- Calls **`simulateRequest`** for each request
- Counts number of hits and misses that occur using **`simulateRequest`** return value
- Returns total number of hits and total number of misses

simulateRequest

```
# -----------------------------------------------------------------------------
# simulateRequest:
#
# Simulates a single request, given a pointer to a request in the REQUESTS
# array. First, simulateRequest must call getTag, getIndex, and getOffset to
# get the tag, index, and offset of the requested address.
#
# Then, simulateRequest must determine if the request is a hit or a miss by
# comparing the tag of the requested address with the tags of the valid blocks
# in the set at the index of the requested address. If block 0 is valid (the
# valid0 bit is 1) and the tag of block 0 matches the tag of the requested
# address, then there is a hit on block 0. Similarly, if block 1 is valid
# (the valid1 bit is 1) and the tag of block 1 matches the tag of the
# requested address, then there is a hit on block 1. Otherwise, the request
# results in a miss.
#
# If the request results in a hit, simulateRequest must call simulateHit and
# return 1. If the request results in a miss, simulateRequest must call
# simulateMiss and return 0.
#
# Args:
#   a0: pointer to a request in the requests array
#
# Returns:
#   a0: 1 if the request resulted in a hit, 0 if the request resulted in a miss
#
# Register usage:
#   insert your register usage for this function here
# -----------------------------------------------------------------------------
```

- Determines if request is a hit or miss
- Calls **`simulateHit`** or **`simulateMiss`** accordingly
- Returns 1 if request resulted in a hit, 0 if request resulted in a miss

simulateHit

```
# ---------------------------------------------------------------------------------
# simulateHit:
#
# Given the block that the hit occurred on (block 0 or block 1),
# simulateHit must set the LRU bit to the opposite block (if the hit was on
# block 0, the LRU bit should be set to 1; if the hit was on block 1, then the
# LRU bit should be set to 0).
#
# If the request was a load, the dirty bit of the requested block must be left
# unchanged. If the request was a store, then simulateHit must set the dirty
# bit of the requested block to 1.
#
# simulateHit must call logHit to print information about the hit.
#
# Args:
#   a0: 0 if the request is a load, 1 if the request is a store
#   a1: tag of the requested address
#   a2: index of the requested address
#   a3: offset of the requested address
#   a4: pointer to meta_bytes[index]
#   a5: 0 if the hit is on block 0, 1 if the hit is on block 1
#
# Register usage:
#   insert your register usage for this function here
# ---------------------------------------------------------------------------------
```

- Simulates a cache hit
- Updates LRU bit
- Sets dirty bit to 1 if request was a store
- Calls **`logHit`** helper function to print information about the hit

simulateMiss

```
# ----------------------------------------------------------------------------
# simulateMiss:
#
# simulateMiss must determine which block to replace by checking the LRU bit
# of the set (if the LRU bit is 0, replace block 0; if the LRU bit is 1,
# replace block 1). Then, simulateMiss must check the valid bit and dirty bit
# of the replaced block to determine if a write-back should occur (if the
# valid bit is 1 and the dirty bit is 1, then write-back; otherwise, don't
# write-back).
#
# simulateMiss must set the dirty bit of the replaced block to
# 0 if the request is a load or 1 if the request is a store.
#
# simulateMiss must call logMiss to print information about the miss and
# whether or not a write-back occurred.
#
# simulateMiss must replace the tag of the replaced block with the tag of the
# request, and set the valid bit of the replaced block to 1. Lastly,
# simulateMiss must update the LRU bit of the set to the non-replaced block
# (if block 0 was replaced, then the LRU bit should be set to 1; if block 1
# was replaced, then the LRU bit should be set to 0).
#
# Args:
#   a0: 0 if the request is a load, 1 if the request is a store
#   a1: tag of the requested address
#   a2: index of the requested address
#   a3: offset of the requested address
#   a4: pointer to tags[index]
#   a5: pointer to meta_bytes[index]
#
# Register usage:
#   insert your register usage for this function here
# ----------------------------------------------------------------------------
```

- Simulates a cache miss
- Checks LRU bit to determine which block to replace
- Checks valid and dirty bit of replaced block to determine if write-back occurs
- Calls **`logMiss`** helper function to print information about the miss
- Sets dirty bit of replaced block to 0 if request was load, 1 if request was store
- Replaces tag of block with tag of request
- Sets valid bit of replaced block to 1
- Updates LRU bit

Recommended lab completion flow

1. Implement **`simulateCache`**
    - Not too hard, just a loop over the **`REQUESTS`** array
2. Implement **`simulateRequest`**
    - A bit more difficult, requires checking the tags in the set at the index of the requested address to determine if hit or miss
3. Implement **`simulateHit`**
    - Simpler than **`simulateMiss`**, so implement it first
4. Implement **`simulateMiss`** last
    - Most complicated part of the lab (will take the most time)

Tips for completing the lab

- **Carefully read the function headers! This will save you time!**
  - This includes the functions you have to implement, as well as helper functions
- Use the helper functions:
  - `logHit` for printing information about a cache hit
  - `logMiss` for printing information about a cache miss
  - `getTag` for getting the tag of an address
  - `getIndex` for getting the index of an address
  - `getOffset` for getting the offset of an address
  - A correct lab solution **must** call `logHit` and `logMiss` to print required output
- Don't forget to fill out the register usage section in the header of all four functions that you implement

Data Structures

`requests` array

- Contains information about each request passed as input
  - Initialized by `common.s`
- `REQUESTS` global variable points to base address of the array
  - `la t0, REQUESTS`
- Each request is represented by two 32-bit words:
  - 0 if the request is a load,
    1 if the request is a store
  - The requested address

Example: `load_and_store.txt` test case consists of two requests:

```
l 0x1771F984
s 0x32BBEF44
```

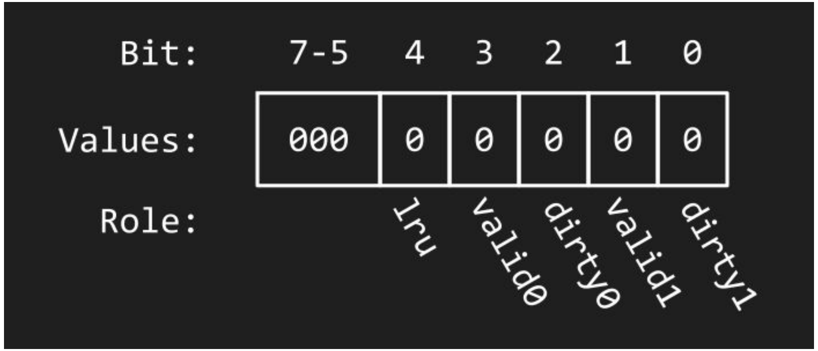
Assuming `REQUESTS = 0x10010000`, the requests array would look like this:

`tags` array

- Contains tags for the blocks in each set
  - Initialized to 0 by `common.s`
- `TAGS` global variable points to base address of the array
  - `la t0, TAGS`
- Each element `tags[i]` of the `tags` array consists of two 32-bit words:
  - Tag of block 0 in the set with index `i`
  - Tag of block 1 in the set with index `i`
- Lab solution must update the tags in the `tags` array when simulating a cache miss

`meta_bytes` array

- Contains LRU bits for each set, and valid and dirty bits for each block
  - Initialized to 0 by `common.s`
- `META_BYTES` global variable points to base address of the array
  - `la t0, META_BYTES`
- Each element of meta bytes array is a meta byte, where:
  - Bits 7-5 are not used
  - Bit 4 is the LRU bit for the set
  - Bit 3 is the valid bit for block 0
  - Bit 2 is the dirty bit for block 0
  - Bit 1 is the valid bit for block 1
  - Bit 0 is the dirty bit for block 1



Testing and Final Remarks

Test case format

- A small number of tests are provided in the Tests directory
  - The provided tests are not extensive: **you should create your own tests to ensure that your solution is correct**
- Expected output for the test files is given in the `*.out` files
- The format of the test input (`*.txt`) files is as follows:

```
[N (# of requests)] [S (# of sets)]
[# of tag bits] [# of index bits] [# of offset bits]
[request 1]
[request 2]
⋮
[request N]
```

Running a test

- To run a test, provide test file as a program argument to RARS
    - Program arguments must be enabled using **pa** flag if running test via terminal, or RARS graphical interface if running test with RARS
- *Example*: To run the **`load_and_store.txt`** test case:

```
rars cacheSimulator.s nc pa Tests/load_and_store.txt
```

Or, using the RARS graphical interface, enter the path in the program arguments field before assembling and running the program:

Final Remarks

- **Read the lab description carefully!**
  - **Watch the videos in the lab description!**
- Create your own test cases for edge cases
  - Pay attention to the Testing section and the Assumptions and Notes section in the lab description to create valid test cases
- Read the marksheet (link in the Marking Guide section of lab description) to see how your lab will be graded
- Don’t be afraid to ask questions
- Style marks are an easy 20%