Due date: Thursday, May 6 11:59:59 PM ET

Points: This lab is worth 12 points (out of 200 points in 6.004).

Collaboration policy: Our full Academic Honesty policy can be found on the Course Information page of our website. As a reminder, **all 6.004 labs should be completed individually**. You may discuss the lab at a high level with a classmate, but you may not work on code together or share any of your code. You must complete the collaboration questions found on the Labs page after turning in your lab and before your checkoff meeting to help ensure that you are abiding by our academic honesty policy.

Prelab: To prepare for the lab, we recommend you complete the corresponding [Prelab Questions] before starting (see: https://6004.mit.edu/web/spring21/labs). Please refer to the "Help" dropdown menu above if you have any questions.

The Prelab Questions must all be answered correctly before you can receive a check-off.

Getting started: To create your initial Lab 7 repository, please either click [Create Repository] on the Labs tab of the class site or visit the Lab 7 Didit page (our lab grading system). All files you need for the lab (except this handout) will be placed in your repository.

Once your repository has been created, you can clone it anywhere in your Athena locker by running the "git clone" command displayed by Didit. This command automatically configures your local repo to print a feedback message from Didit on your Terminal when you make a submission.

Alternatively, you can clone a local repo without the feedback message feature by running the `git clone` command in your terminal:

```
git clone git@github.mit.edu:6004-sp21/lab7-{YourMITUsername}.git
```

Turning in the lab: To turn in this lab, commit and push the changes you made to your git repository. You can submit as many times as you like. **After pushing, check Didit (https://didit.mit.edu/6.004/sp21/ →Current Lab →Your Kerberos Name →Latest Build #) to verify that your submission was correctly pushed and passes all mandatory tests.** If you finish the lab in time but forget to push, you will incur the standard late submission penalties. After pushing the lab, you must also complete the collaboration questions for this lab found on the Labs tab of the course website.

Didit may take up to 5 minutes to compile and test code that is pushed to your repository. Therefore, it is more efficient to test locally for faster feedback. Once your code passes local tests and you've pushed your changes, check that the commit appears in Didit and that it was graded properly to receive credit.

Check-off meeting: After turning in this lab, you must **sign up** for and do a check-off by Wednesday, May 12th. The checkoffs for this lab will begin on Friday, April 30th. See the course website for check-off hours/instructions.

To complete this lab you must complete and PASS all of the exercises, and answer all the discussion questions. Remember that you cannot pass 6.004 unless you have completed ALL labs, including this one.

Introduction

In this lab you will implement two caches: a *direct-mapped cache*, and a *two-way set-associative cache* with the LRU (least-recently used) replacement policy.

Figure 1 shows the overall structure of direct-mapped and two-way set-associative caches, and illustrates how a lookup is performed in each cache, as we saw in Lecture 16.

A direct-mapped cache (Figure 1a) has three memory arrays. First, the *data array* stores data in blocks of multiple consecutive words, called *cache lines*. Second, the *tag array* stores the *tag* of each cache line, i.e., the bits of its address needed to uniquely identify its memory location. Third, the *status array* holds valid and dirty bits for each cache line. In a direct-mapped cache, each line can reside in a single location in the cache, i.e., a single row of the memory arrays.

To perform a lookup in a direct-mapped cache, we divide the address into three fields: the *word offset bits* identify the particular word accessed within the cache line; the *index bits* determine the location (row) in the cache where the line can reside; and the *tag bits* are all the remaining bits. A lookup first reads the status, tag, and data arrays at the location given by the index bits. If the location has a valid line and the stored tag matches the tag bits, we have a *cache hit*, and the data is served from the data array (using the offset bits to select the right word). Otherwise, the lookup results in a *cache miss* and data is fetched from main memory.

Set-associative caches (Figure 1b) reduce misses by allowing each address to reside in one of multiple locations. An N-way set-associative cache can be seen as N replicas of a direct-mapped cache. We call each such replica a *way*. Each line can reside in a single location (row) of each way, so it maps to one of N possible locations across the cache. We call this group of locations (a row across all ways) a *set*.

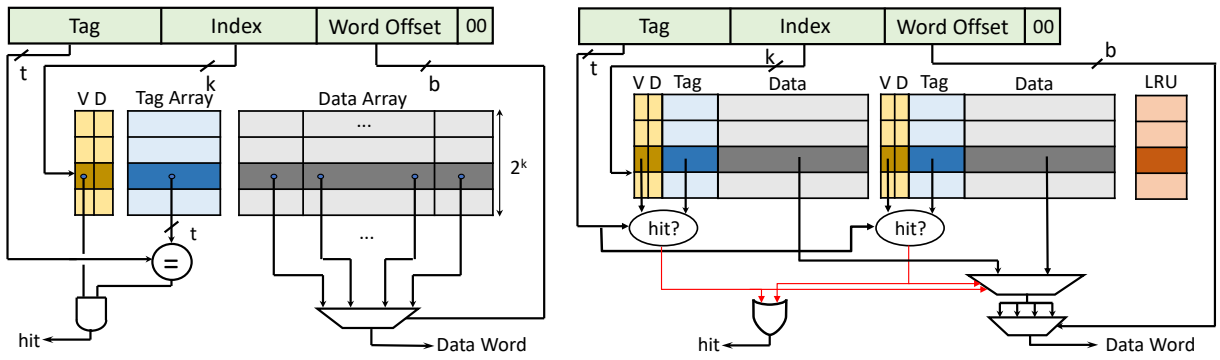
A lookup in a set-associative cache checks all the ways in parallel. A cache hit happens when one of the ways has a valid line with a matching tag. If there are no matches, the cache fetches the line from memory and selects which of the cached lines to replace. The cache implements a *replacement policy* for this purpose. In our case, the two-way set-associative cache will use the LRU replacement policy, which maintains an *LRU array* that tracks which of the two lines in each set was accessed least recently.

Coding guidelines: You are only allowed to make changes to **`CacheHelpers.ms`**, **`DirectMappedCache.ms`**, and **`TwoWayCache.ms`**. Modifications to other files will be overwritten during Didit grading.

Discussion questions: The discussion questions in this lab are worth **5%** of your grade. Please write your answers in the `discussion_questions.txt` file. You can update your answers before your checkoff meeting, but you must submit an initial answer to each question when you submit the lab. You should be prepared to explain your answers during the checkoff.

1 Cache Implementation Overview

To guide your implementation, this section presents *(i)* the *interface* that your cache should follow, i.e., its input and methods; *(ii)* the modules that your cache should use as *building blocks*, i.e., SRAM arrays and main memory; and *(iii)* the *finite-state machine* that your cache should implement to perform accesses.



(a) Direct-mapped cache

(b) Two-way set-associative cache

Figure 1: Structure of direct-mapped and two-way set-associative caches.

Cache interface: Figure 2 details the main interface of each cache, both in Minispec code and graphically. Both the direct-mapped and two-way caches use exactly the same interface.

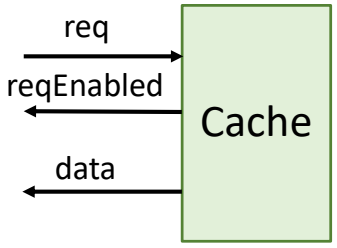
```
typedef enum {Ld, St} MemOp;
typedef Bit#(32) Word;
typedef struct {
   MemOp op;
   Word addr;
   Word data;
} MemReq;
```

(a) Memory request struct

```
module Cache;
  input Maybe#(MemReq) req
        default = Invalid;
  method Bool reqEnabled;

  method Maybe#(Word) data;
endmodule
```

(b) Cache input and methods



(c) Cache circuit

Figure 2: Cache interface: memory request, inputs and methods, and equivalent cache circuit.

The cache has a single input, `req`, and two outputs, `reqEnabled` and `data`. `req` is an optional memory request (a `Maybe#(MemReq)`). When `req` is valid, the request has three fields, as shown in Figure 2a:

- `op` is the operation, which can be a load (`Ld`) or a store (`St`).
- `addr` is the 32-bit (byte) address.
- `data` is the data that should be written if the request is a store; on a load, the `data` field is ignored.

When `req` is a valid input, the cache should start processing the memory request. Each request will take multiple cycles to perform: at least one cycle if it's a hit, and more cycles on a miss. While serving the current request, the cache needs to prevent its user from giving it another request. This means that the cache only processes one request at a time; this is called a *blocking* cache design, because the cache blocks requests while it's busy. The cache's `reqEnabled` output achieves this purpose: if `reqEnabled` is `True`, the cache is ready to accept a new request, and the cache's user can give a new request by setting a valid `req` input. Otherwise, the cache is busy and the cache's user must not give a valid `req` input. Finally, if the request is a load, the cache will output the requested word as a valid `Word` through the `data` method. The cache will output the data for one cycle, and assumes that the cache's user will always read it, so it does not need to continue to output the read value for multiple cycles.

Note how the cache interface has two key differences from the multi-cycle sequential modules we've seen so far (e.g., the folded multiplier from Lab 5 or the examples from the Minispec sequential tutorial). First, in prior modules, the module user could always give a valid input, which would interrupt the current computation; but here, the cache can prevent its user from giving a valid input. This is because interrupting a memory request is both extremely complicated, and unnecessary. Second, in prior modules, the module would hold the result until a new computation was started, but here, the cache outputs the data for a single cycle. Again, this simplifies the design by avoiding an unnecessary register within the cache.

In addition to this interface, each cache should implement two methods, `getHits` and `getMisses`. These methods are not shown in Figure 2. They return the hit and miss counts and are used by tests to check that each cache behaves as expected. The skeleton code already provides these methods; you only need to increment their corresponding hit and miss counters when appropriate.

Cache building blocks: Each cache uses two main types of modules. First, the cache interfaces with a slower *main memory*, which is typically implemented with DRAM. Second, the cache implements the tag, status, and data arrays using *SRAM memories*. Figure 3 sketches the implementation of a direct-mapped cache, showing how it uses these modules.

Figure 3 shows that main memory is considered internal to the cache. This means the cache controls main memory, and the cache's user does not have to deal with the connection between cache and main memory. (In the code, the main memory is passed as an argument to the cache module; this will be useful with the pipelined processor, so its instruction and data caches can share the main memory).

Figure 3 also illustrates the main memory interface (defined in `MainMemory.ms`). This interface is *nearly identical* to the cache's interface: there is a `req` input and `reqEnabled` and `data` outputs with the same semantics (a valid `req` initiates a new memory request, `reqEnabled` denotes whether main memory is ready to take in a request, and `data` outputs valid data on a load request). However, there is a key difference:

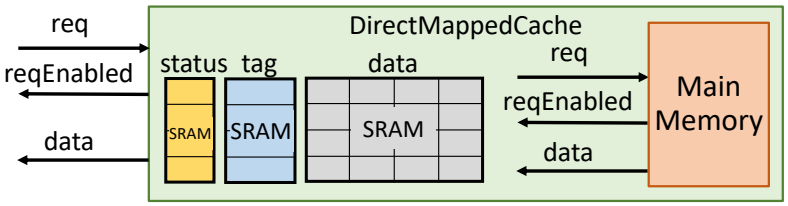


Figure 3: Implementation sketch of the direct-mapped cache, showing the SRAM arrays and the main memory interface.

```
typedef Vector#(16, Word) Line;
typedef Bit#(26) LineAddr;
typedef struct {
   MemOp op;
   LineAddr lineAddr;
   Line data;
} LineReq;
```

Figure 4: Main memory data, address, and request types.

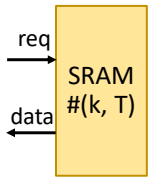
main memory loads and stores *entire cache lines* instead of 32-bit words. Figure 4 shows the `Line` type, which is a 16-word vector (i.e., we will use 64-byte cache lines), and the `LineReq` type. In `MainMemory`, `req` is a `Maybe#(LineReq)` input, and `data` returns a `Maybe#(Line)`.

```
typedef struct {
   Bit#(k) addr;
   Bool write;
   T data;
} SRAMReq#(k,T);
```

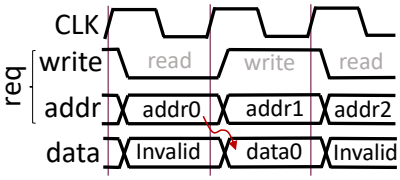
(a) SRAM request

```
module SRAM#(Integer k, type T);
   input Maybe#(SRAMReq#(k,T))
      req default = Invalid;
   method Maybe#(T) data;
endmodule
```

(b) SRAM input and methods



(c) SRAM circuit



(d) SRAM timing diagram

Figure 5: SRAM interface: request type, inputs and methods, equivalent circuit, and timing.

Figure 5 details the interface of SRAM memory arrays (available in `SRAM.ms`). `SRAM` is a parametric module: `SRAM#(k, T)` is a memory of 2^k entries of type `T`. SRAMs have a simpler interface than caches and main memory: they have a `req` input and a `data` method, but no `reqEnabled` method. This is because SRAMs are *non-blocking*: they can take a new memory request every cycle. SRAMs are also *single-cycle*, as shown in Figure 5d: when the request is a read (i.e., the request's `write` field is `False`), the SRAM returns its corresponding data *on the next cycle*, as a valid result of the `data` method. For example, in Figure 5d, the read of address `addr0` in cycle 0 causes its corresponding data, `data0`, to be returned on cycle 1.

Cache request FSM: Handling a cahe request may entail one or more actions spread over multiple cycles. Let's consider the different options. Every request begins with a *lookup*, a read of the tag, status, and data arrays. Depending on the values that these arrays return, there are three possible cases. In order of increasing complexity, these cases are:

1. *Hit:* On a hit, the cache simply needs to return one of the words read from the data array (if it's a load) or update the data array with the written data (if it's a store). On a store, the cache should also mark the line as dirty.
2. *Clean miss:* If there is a miss and the line we need to replace is clean (i.e., not dirty), then the cache does not need to write back the replaced line. The cache must request the missing line from main memory. Once the line is read, the cache must fill the tag/data/status arrays with the new line; this operation is called a *fill*. In addition, a fill must perform the operations in a hit: serving the data if the request is a load, or updating the line before writing it to the data array and marking the line dirty if the request is a store.
3. *Dirty miss:* If there is a miss and the line we need to replace is dirty, then the cache must first write back the line to main memory, then request the missing line from main memory and perform the fill like before. Note that, to serve a dirty miss, the cache needs to perform two main memory requests, not one as in a clean miss.

We recommend that you handle these cases by implementing the 4-state FSM shown in Figure 6. (Other FSMs are possible, and some have higher performance, but this option is simple and sufficient.)

The four states in Figure 6 handle requests as follows:

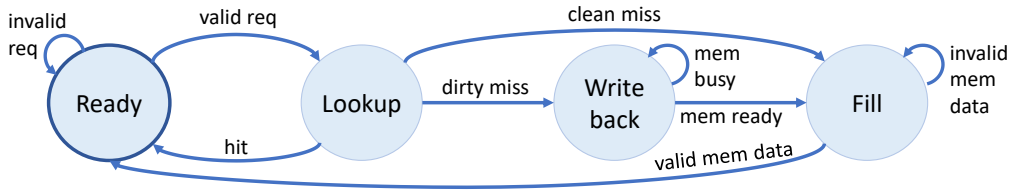


Figure 6: State-transition diagram for handling a cache request.

1. *Ready:* A cache in the `Ready` state is ready to accept a new request. Upon receiving a new request (i.e., a valid `req` input), it requests a read to the proper index of the tag/status/data SRAM arrays, and transitions to the `Lookup` state.
2. *Lookup:* In the `Lookup` state, the tag/status/data SRAM arrays output the data that was requested in the previous cycle (while in `Ready`). Based on this data, the cache determines whether it has a hit, a clean miss, or a dirty miss. Upon a hit, the cache serves (if load request) or updates (if store request) the relevant data, and transitions back to the `Ready` state. Upon a clean miss, the cache requests the missing line from main memory, and transitions to the `Fill` state. Upon a dirty miss, the cache writes back the replaced line to main memory, and transitions to the `Writeback` state.
3. *Writeback:* In the `Writeback` state, the cache has issued a write-back request to main memory, and is waiting for the memory to handle that request. The cache stays in the `Writeback` state for as long as the memory is busy. Once main memory is ready to take a new request (i.e., `mainMem.reqEnabled == True`), the cache requests the missing line from main memory, and transitions to the `Fill` state.
4. *Fill:* In the `Fill` state, the cache has requested the missing line from main memory, and is waiting for memory to return the data. The cache stays in the `Fill` state until main memory returns the line. Then, the cache fills the relevant tag/status/data arrays with the line (updating it if store request), serves the requested word at its own output (if load request), and transitions back to the `Ready` state.

2 Extracting the Tag, Index, and Offset Bits

Your first task is to write three functions that extract the *index*, *word offset*, and *tag* fields from a memory address, as described in the introduction. Your cache implementation should then use this functions, which will be simpler than extracting these fields directly.

Your functions should use two `Integer` constants (defined in `CacheTypes.ms`): `logCacheSets` is the logarithm base-2 of the number of cache sets, and `logWordsPerLine` is logarithm base-2 of the number of words per line. We will implement caches with 64 sets and 16 words per line, so these constants are 6 and 4, but if you use the constants instead of fixed numbers, you'll be able to change your cache size later on.

Exercise 1 (10%): Implement the helper functions that extract the cache line word offset, index, tag fields, and line address (tag + index) from a given memory address in `CacheHelpers.ms`.

Test your implementations by running `make HelperTest`

3 Direct-Mapped Cache

Before implementing the direct-mapped cache, consider the following sequence of memory accesses:

```
Store at address  0x00BC
Load from address 0x00BC
Load from address 0x1094
Store at address  0x0084
Load from address 0x00BC
```

This sequence is part of the cache tests, so it is helpful to understand how the cache should behave on it.

Discussion Question 1 (1%): What are the index, line offset, and tag bits for each of the requests above?

Discussion Question 2 (2%): What is the sequence of states that the cache FSM goes through when processing this sequence of requests? You may assume that the lines in the cache are initially not valid.

Now complete `DirectMappedCache.ms` in order to implement a direct-mapped cache. You will need to specify what happens in each of the four possible cache states, as indicated in the skeleton code.

Exercise 2 (45%): Implement a direct-mapped cache by completing the `DirectMappedCache` module in `DirectMappedCache.ms`. Note that you should also keep track of hit and miss counts.

You can find all necessary type definitions in `CacheTypes.ms`.

Hint 1: Although each SRAM array returns data in a Maybe type, in most cases your cache already knows whether the data is valid. For instance, in the `Lookup` state, the data is always valid because the previous state (`Ready`) always issues a read to all arrays. In these cases, it's simpler to not check the valid bits, and simply get the data with `fromMaybe` (e.g., `let tag = fromMaybe(?, tagArray.data);`).

Hint 2: You can assume that main memory is ready to take in a new request (i.e., `mainMem.reqEnabled == True`) if the cache doesn't have a pending request to it. In particular, this means that in the `Lookup` state, on a miss, you do not need to wait for memory to become ready and can issue a request immediately (this is why in Figure 6 there's no transition in `Lookup` to handle a busy memory).

Hint 3: You can use the following syntax to initialize a struct in Minispec:

```
LineReq req = LineReq{op: St, lineAddr: dummyLineAddr, data: dummyData};
```

In addition, note that we have provided the following type synonyms in the template code which may prove useful:

```
typedef SRAMReq#(logCacheSets, CacheTag) TagReq;
typedef SRAMReq#(logCacheSets, Line) DataReq;
typedef SRAMReq#(logCacheSets, CacheStatus) StatusReq;
```

We provide two test suites for your cache. First, the *microtests* use a short, carefully chosen sequence of accesses to exercise most of the cache. Second, the *Beveren* randomized test uses a much longer sequence that tests the cache more thoroughly. We recommend that you first run the microtests, then the Beveren test only when the microtests pass.

Passing the microtests alone but not the Beveren test is worth 25% (instead of 45%) for this exercise. However, you also need to pass Beveren to pass the lab.

To test your design on the microtests, run `make DirectMappedMicrotest`

To test your design on Beveren, run `make DirectMappedBeveren`

4 Two-way Set-Associative Cache

Before implementing the two-way set-associative cache (as shown in Figure 1b), consider the same sequence of memory access as in the previous discussion questions:

```
Store at address  0x00BC
Load from address 0x00BC
Load from address 0x1094
Store at address  0x0084
Load from address 0x00BC
```

Discussion Question 3 (2%): Consider the two-way set-associative cache. What is the sequence of states that the cache FSM goes through when processing this sequence of requests? You may assume that the lines in the cache are initially invalid.

Exercise 3 (40%): Implement the two-way set-associative cache by completing the `TwoWayCache` module in `TwoWayCache.ms` using a Least-Recently-Used (LRU) cache replacement policy. Note that you should also keep track of hit and miss counts.

Hint: The code for the two-way cache has many similarities with the direct-mapped cache. You may want to start by copying your DirectMappedCache code and modifying it to use multiple ways. The main modifications are:

1. On a lookup, you should read all arrays in parallel and search all of them to find a hit.
2. On every access, you should keep track of the least-recently accessed line in the LRU array.
3. On a miss, you should use the LRU array to find which line to evict.

Just like before, the two-way cache has two kinds of tests, *microtests* and the *Beveren* randomized test. Passing the microtests alone but not the Beveren test is worth 20% (instead of 40%) for this exercise. However, you also need to pass Beveren to pass the lab.

To test your design on the microtests, run `make TwoWayMicrotest`

To test your design on Beveren, run `make TwoWayBeveren`