Due date: Thursday 4/18 11:59:59pm EDT.

Getting started: To create your initial Lab 7 repository, please visit the repository creation page at https://6004.mit.edu/web/spring19/user/labs/lab7. Once your repository is created, you can clone it into your VM by running:

```
git clone git@github.mit.edu:6004-spring19/labs-lab7-{YourMITUsername}.git lab7
```

Turning in the lab: To turn in this lab, commit and push the changes you made to your git repository. After pushing, check the course website to verify that your submission passes all the tests. If you finish the lab in time but forget to push, you will incur the standard late submission penalties.

Check-off meeting: After turning in this lab, you are required to go to the lab for a check-off meeting by Wednesday 4/24. See the course website for lab hours.

1 Caches

In this Lab you will implement a Direct-Mapped Cache and a Two-Way Set-Associative Cache using LRU replacement policy. Lecture 15 gives an overview of what caches are, the motivations behind using them, and how the different types of caches operate.

For this Lab you must complete and PASS all of the Exercises

Coding guidelines: You are only allowed to change the following files: CacheHelpers.bsv, CAU.bsv, DirectMappedCache.bsv, TwoWayCache.bsv and FixedReqTest.bsv. Modifications to other files will be overwritten during didit grading.

2 Cache Line Basics

As discussed in lecture, data words are stored in caches in what are known as cache lines. The value of the memory address of a given data word is used to calculate which line of the cache that word will be stored in. This is done by extracting the index, word offset, and tag of the target cache line from the memory address. In this lab, the code will make use of three helper functions that will extract these fields from a memory address. CacheTypes.bsv contains some important cache-related type declarations, such as the number of cache lines and words per line, that you will need in order to correctly implement these functions.

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

Test your implementations by running `make HelperTest; ./HelperTest`
3 Caches in Action

Imagine you have a cache that is Direct-Mapped and has 4 lines with 4 data words per line. Given the following sequence of memory address accesses, answer the discussion questions regarding the state of the cache.

Memory address sequence:
0x1040, 0x1054, 0x105C, 0x246C, 0x25F0, 0x3334, 0x104C, 0x25F8, 0x201C, 0x1058

**Discussion Question 1:** What is the tag of each cache line at the end of the memory access sequence? What is the corresponding address of the 1st data word in each line?

**Discussion Question 2:** How many hits and misses will the memory access sequence incur in the cache? How would changing the cache to Two-Way Set Associative with LRU replacement affect these amounts?

4 Cache-Array Unit

In this lab you will construct your Direct-Mapped Cache and your Two-Way Set Associative Cache using a Cache-Array Unit (CAU) implemented in Bluespec. The CAU encapsulates the data, status, and tag arrays of a cache, which are all made using SRAM. In particular, the CAU should accept requests from the Cache module for a given Memory address, make requests to the various arrays for that address, decide whether the request results in a hit or miss, and respond to the cache in kind. Lecture 16 gives further details about the implementation of the CAU as well as both types of caches you will implement.

The **Appendix** contains several important enum and struct definitions from across the various source files in a single place for your convenience.

**Exercise 2 (20 Points):** Implement the CAU by completing the skeleton code in CAU.bsv. Test your design by running `make CAUTest; ./CAUTest`.

5 Direct-Mapped Cache

**Exercise 3 (30 Points):** Implement a blocking direct-mapped cache by completing the skeleton code in DirectMappedCache.bsv using the CAU you implemented. To test your design with Beveren test, run `make BeverenDirectMap; ./BeverenDirectMap`. To compile your design with a Multi-cycle Processor, run `make MultiCycleDirectMap; ./MultiCycleDirectMap`. You can run a suite of tests on the processor using the `test.sh` by running `./test.sh` or `bash test.sh`. You should pass the microtests (option 5) and the fullasmtests (option 6).

6 Two-way Set-Associative Cache

**Exercise 4 (30 Points):** Implement a blocking two-way set-associative cache by completing the skeleton code in TwoWayCache.bsv using a Least-Recently-Used (LRU) cache replacement policy. To test your design with Beveren test, run `make BeverenTwoWay; ./BeverenTwoWay`. To compile your design with a Multi-cycle Processor, run `make MultiCycleTwoWay; ./MultiCycleTwoWay`. You can run a suite of tests on the processor using the `test.sh` by running `./test.sh` or `bash test.sh`. You should pass the microtests (option 5) and the fullasmtests (option 6).
7 Cache Hit and Miss

Exercise 5 (10 points): Implement `getHitCount` and `getMissCount` methods in `DirectMappedCache.bsv` and `TwoWayCache.bsv`. The `getHitCount` and `getMissCount` methods are a part of the `Cache` interface for getting hit and miss counts from the cache.

```haskell
interface Cache#(numeric type logNumLines);
  // methods for the processor to interact with the cache
  method Action req(MemReq req);
  method ActionValue#(Word) resp();
  // methods for the cache to interact with DRAM
  method ActionValue#(LineReq) lineReq;
  method Action lineResp(Line r);
  // methods for getting the cache hit and miss counts
  method Bit#(32) getHitCount;
  method Bit#(32) getMissCount;
endinterface
```

You should add a cache hit counter and a cache miss counter for each of the Direct-Mapped Cache and the Two-Way Set-Associative Cache. Each counter should be a 32-bit register, and each counter should only be incremented in one rule in your cache.

Test your design by running `make HitMissTestDirectMap; ./HitMissTestDirectMap` and `make HitMissTestTwoWay; ./HitMissTestTwoWay`.

Now that we have access to hit and miss counts of our cache, we can examine the cache more.

Fill in the test vector in `FixedReqTest.bsv` to produce two tests that show you have a Two-Way Set-Associative Cache and that the replacement policy is LRU respectively. The tests will print the number of cache hits and misses for your cache. Solely from the hit and miss counts, you should be able to show that your cache is a 2-way set associative cache and NOT any other type of associativity using the first test vector, and that your cache uses an LRU replacement policy and NOT any other type of replacement policy (such as MRU, random, first item, etc.) from the second test vector.

Test your designs by running `make FixedReqTest; ./FixedReqTest`.

Discussion Question 3: What is your test vector for testing that your cache is 2-way set associative? How does your run of `FixedReqTest` show that your cache is 2-way set associative?

Discussion Question 4: What is your test vector for testing that your cache uses the LRU replacement policy? How does your run of `FixedReqTest` show that your cache is implementing the LRU replacement policy? Does your vector make you reasonably sure (more than 50%) that the replacement policy is truly LRU and not random with decisions that appear to be LRU?
8 Cache-Efficiency of Software

Caches are critical to the performance of processors as they help avoid the huge latencies of main memory, while only using a small amount of expensive SRAM. However, while having a great cache can give huge performance improvements, it’s equally important that your software is written in a way that takes advantage of your particular cache. We ran two different implementations of a Matrix Multiplication program on a processor using a Two-Way Set Associative cache with LRU replacement, 16 words per line, and 16 lines. Both versions of the program are below, with matrix_multiply_good being more “cache-efficient” than matrix_multiply_bad.

```c
void matrix_multiply_good(int a[N][N], int b[N][N], int c[N][N]) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++)
                c[i][j] += mult(a[i][k], b[k][j]);
        }
    }
}

void matrix_multiply_bad(int a[N][N], int b[N][N], int c[N][N]) {
    int i, j, k;
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++)
            for (i = 0; i < N; i++)
                c[i][j] += mult(a[i][k], b[k][j]);
    }
}
```

The cache-efficient version had 33,985,902 cache requests with 17,193,669 misses for a 50.6% miss rate, while the cache-inefficient version had 33,920,364 cache requests with 33,644,740 misses for a 99.2% miss rate.

**Discussion Question 5:** What is the difference in the memory access patterns between the two code versions? Draw what these patterns look like in matrices A, B, and C.

**Discussion Question 6:** Why does this difference in memory access patterns affect the cache miss rate of the code?
9 Appendix

Type Definitions: Listed below are the definitions of various enums and structs from the different source files.

**HitMissType** reports whether a request to the CAU resulted in a *Load Hit*, *Store Hit*, or *Miss*.

```haskell
typedef enum {
    LdHit,
    StHit,
    Miss
} HitMissType deriving (Bits, Eq, FShow);
```

The **CAUStatus** gives the current state of the blocking CAU, *Busy* or *Ready*.

```haskell
typedef enum {
    Ready,
    Busy
} CAUStatus deriving(Eq,FShow,Bits);
```

The **CacheStatus** represents whether a given cache line is currently *Invalid*, *Clean*, or *Dirty*.

```haskell
typedef enum {
    Invalid = 0,
    Clean = 1,
    Dirty = 2
} CacheStatus deriving (Bits, Eq, FShow);
```

The **ReqStatus** in the direct-mapped cache and two-way cache dictate which stage the currently executing request is in within the cache.

```haskell
typedef enum {
    Ready,
    WaitCAUResp,
    SendReq,
    WaitDramResp
} ReqStatus deriving(Bits, Eq, FShow);
```

**TaggedLine** is a struct containing the data, tag, and status of a single cache line.

```haskell
typedef struct {
    Line line;
    CacheStatus status;
    CacheTag tag;
} TaggedLine deriving {Bits, Eq, FShow};
```
CAUResp represents the response returned by the CAU, including the hit or miss result, and either the result of a load hit or the value returned from a dirty capacity miss.

```haskell
typedef struct {
    HitMissType hitMiss;
    Word ldValue;
    TaggedLine taggedLine;
} CAUResp deriving {Bits, Eq, FShow};
```

A MemReq struct represents a single Load or Store request to Main Memory.

```haskell
typedef struct {
    MemOp op;
    Word addr;
    Word data;
} MemReq deriving (Bits, Eq, FShow);
```

A LineReq is a line-based memory request, with addresses always being line addresses and the data is a complete cache line.

```haskell
typedef struct {
    MemOp op;
    LineAddr lineAddr;
    Line data;
} LineReq deriving(Bits, Eq, FShow);
```