ECE 4750 Computer Architecture, Fall 2022
Lab 3: Blocking Cache

School of Electrical and Computer Engineering
Cornell University

revision: 2022-11-02-00-49

In this lab, you will design two finite-state-machine (FSM) cache microarchitectures, which we will eventually compose with the processor designs you developed in the previous lab. The baseline design is a direct-mapped, write-back, write-allocate cache, and the alternative design is a two-way set associative cache that should reduce the miss rate by avoiding conflict misses. You are required to implement the baseline and alternative designs, verify the designs using an effective testing strategy, and perform an evaluation comparing the two implementations. **The milestone for this lab is to complete as many transactions as you can for the baseline design along with an initial set of directed tests. You should consult the lab logistics document for more information about the expectations for all lab assignments and how they will be assessed.**

This lab is designed to give you experience with:

- basic memory system design;
- complex finite-state-machine cache controllers;
- microarchitectural techniques for implementing cache associativity;
- abstraction levels including functional- and register-transfer-level modeling;
- design principles including modularity, hierarchy, and encapsulation;
- design patterns including message interfaces, control/datapath split, and FSM control;
- agile design methodologies including incremental development and test-driven development.

This handout assumes that you have read and understand the course tutorials. You should have already used the `ece4750-lab-admin` script to create or join a GitHub group. To get started, login to an `ecelinux` server, source the setup script, and clone your lab group's remote repository from GitHub:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:cornell-ece4750/lab-groupXX
```

where `XX` is your group number. **You should never fork your lab group's remote repository! If you need to work in isolation then use a branch within your lab group's remote repository.** If you have already cloned your lab group's remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% pytest ../lab3_mem
```

All of the tests for the provided functional-level model should pass, while the tests for the baseline and alternative cache designs should fail. For this lab, you will be working in the `lab3_mem` subproject which includes the following files:

• `CacheFL.py`	– FL cache
• `CacheBaseDpath.v`	– Verilog RTL direct-mapped cache's datapath
• `CacheBaseCtrl.v`	– Verilog RTL direct-mapped cache's control unit
• `CacheBase.v`	– Verilog RTL direct-mapped cache
• `CacheBase.py`	– Python wrapper for baseline cache
• `CacheAltDpath.v`	– Verilog RTL set-associative cache's datapath
• `CacheAltCtrl.v`	– Verilog RTL set-associative cache's control unit
• `CacheAlt.v`	– Verilog RTL set-associative cache
• `CacheAlt.py`	– Python wrapper for alternative cache
• `mem-sim`	– Cache simulator for evaluation
• `__init__.py`	– Package setup
• `test/CacheFL_test.py`	– FL cache unit tests
• `test/CacheBase_test.py`	– Direct-mapped cache unit tests
• `test/CacheAlt_test.py`	– Set-associative cache unit tests
• `test/__init__.py`	– Package setup

1. Introduction

Accessing main memory can require hundreds of cycles, but cache memories can significantly reduce the average memory access latency for well-structured address patterns. Caches are faster than main memory because they are smaller and are located close to the processor; but because a cache can only hold a subset of all memory locations at any one time, we must carefully manage what data we keep in the cache. A cache hit occurs when the data we are requesting is already in the cache, while a cache miss occurs when the data we are requesting is not in the cache and thus requires accessing main memory. Caches exploit spatial and temporal locality to increase the number of cache hits. In an address pattern with significant spatial locality, if we access a given address then in the near future, we are likely to access an address close to the first address. In an address pattern with significant temporal locality, if we access a given address then in the near future, we are likely to access that same address again. In this lab, you will implement and evaluate two cache microarchitectures that organize cache lines in two different ways: (1) direct-mapped where every cache line can only be placed in a single location in the cache, and (2) two-way set-associative where every cache line can be placed in one of two locations in the cache. Both caches will use a write-back, write-allocate policy for handling write misses. Additionally, both caches will have the ability to act as a bank in a larger multi-bank cache organization.

We have provided you with a functional-level model of a cache, which essentially just passes all cache requests through to the memory interface, and passes all memory responses through to the cache response interface. While this might not seem useful, the functional-level model will enable us to develop many of our test cases with the test memory before attempting to use these tests with the baseline and alternative designs.

Figure 1 shows a block-level diagram illustrating how the functional-level, baseline, and alternative designs are integrated with a test source, test sink, and test memory for testing and evaluation. We will load data into the test memory before resetting the cache. Once we start the execution, the

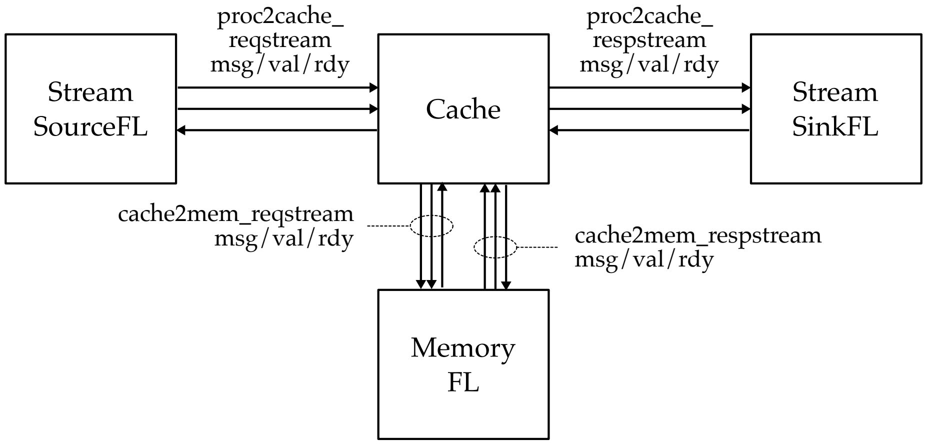
Stream SourceFL
proc2cache_ reqstream msg/val/rdy
Cache
proc2cache_ respstream msg/val/rdy
Stream SinkFL
cache2mem_reqstream msg/val/rdy
cache2mem_respstream msg/val/rdy
Memory FL

Figure 1: Memory System – The cache is integrated with a stream source, stream sink, and memory for testing and evaluation.

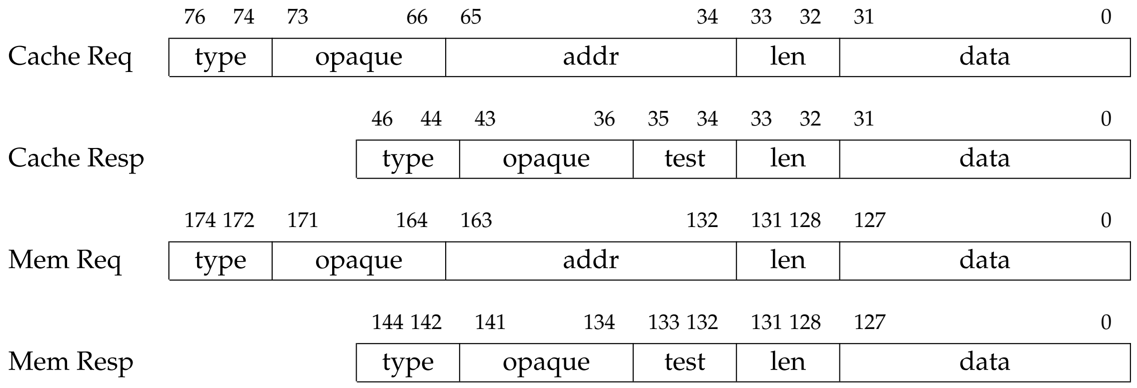

Figure 2: Cache and Memory Request/Response Message Formats – Cache request/response messages are sent between the processor and cache and contain four bytes of data (i.e., one word), while memory request/response messages are sent between the cache and the test memory and contain 16 bytes of data (i.e., one cache line).

test source will send memory requests into the cache, and eventually the cache will send memory responses to the test sink. If the cache needs to access main memory, then the cache will send memory requests to the test memory, and eventually the test memory will send memory responses back to the cache. We make extensive use of the latency insensitive stream interfaces that use the val/rdy micro-protocol in the cache interface. There are four different stream interfaces.

- `proc2cache_reqstream` : processor to cache request stream
- `proc2cache_respstream` : processor to cache response stream
- `cache2mem_reqstream` : cache to memory request stream
- `cache2mem_respstream` : cache to memory response stream

The message formats for memory requests and responses are shown in Figure 2, and the corresponding Verilog `structs` are defined in `vc/mem-msgs.v`. Memory requests use fields to encode the type (e.g., read, write, init), the address, the length of data in bytes, and the data. Memory responses use fields to encode the type (e.g., read, write), the length of data in bytes, and the data. The opaque

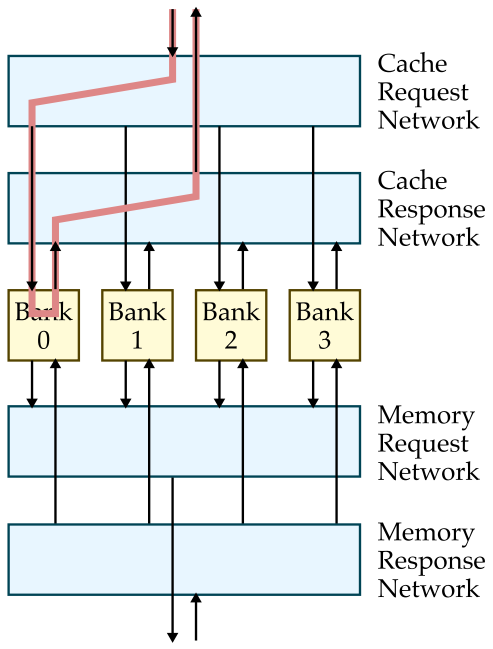

Figure 3: Banked Cache Organization – Four-bank cache organization. Highlighted path is for a memory request that hits in bank 0.

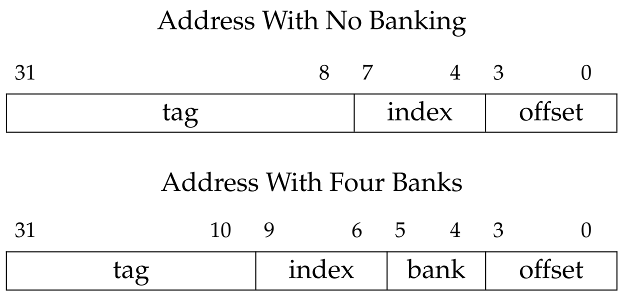

Figure 4: Memory Address Formats With and Without Banking – When using the baseline or alternative design as a bank in a larger cache, we need to exclude the bank bits when indexing into the cache since the request network will take care of routing messages to the correct bank.

field can be used for implementation defined behavior. You should always ensure the opaque field is correctly preserved in the response. Note that the memory messages used for the `cachereq` and `cacheresp` interfaces are for a single word (i.e., 32-bit data field and 2-bit length field), while the memory messages used for the `memreq` and `memresp` interfaces are for an entire cache line (i.e., 128-bit data field and 4-bit length field). If the length field is one then only the least significant byte of the data field (i.e., bits 7-0) is valid. If the length field is two then only the least significant two bytes of the data field (i.e., bits 15-0) are valid. If the length field is zero then all bytes are valid. Note that while the memory message format is quite flexible, our cache designs will only support 4-byte cache requests and 16-byte memory requests. The data field can contain an arbitrary value in a write memory request, however *the data field must contain all zeros in a write memory response*. This simplifies creating reference responses when testing. We add a two-bit test field to each `cacheresp` and `memresp` message. We use the test field in `cacheresp` for testing. If a `cachereq` ends up with a cache miss, we should set the corresponding `cacheresp` message's test field to be `2'b0`. If a `cachereq` turns out to be a cache hit, we should set the corresponding `cacheresp` message's test field to be `2'b1`. By using the test field in the test harness can verify whether a cache transaction is a hit or a miss.

One way to increase cache bandwidth is to enable a cache to process multiple transactions at the same time. Figure 3 shows an alternative approach based on a *banked cache* organization. In a banked cache, we add a request network which directs a cache request to the appropriate bank based on some bits in the address of this cache request. Cache responses are returned over a different response network. Different cache banks can be potentially execute different transactions at the same time, and this increases the overall throughput of the system. The cache you design in this lab will be used both on its own (i.e., in a single-bank configuration) as well as in a four-bank configuration. Figure 4 illustrates which bits are used by the request network to direct a given cache request to the

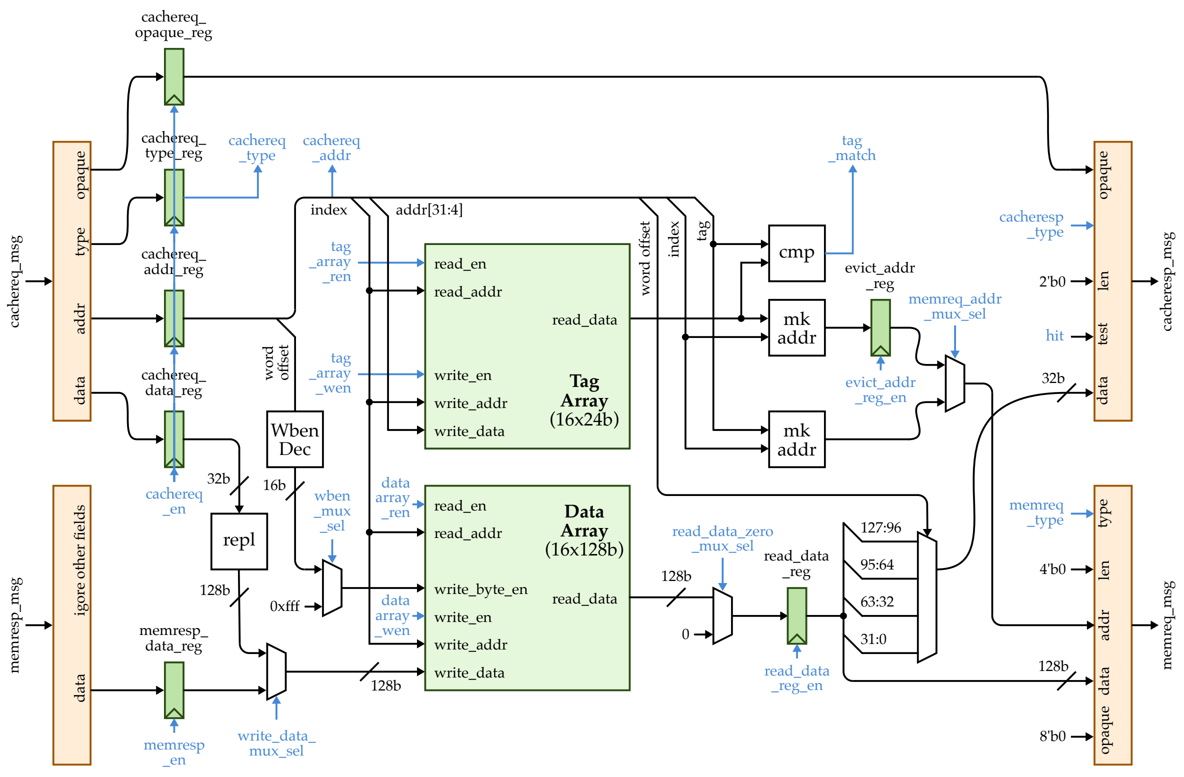

Figure 5: Baseline Datapath – Direct-mapped, write-back, write-allocate cache with 16-byte cache lines and a total capacity of 256 bytes. repl = replicate 32b four times to create 128b signal; mkaddr = concatenate address as appropriate. Orange blocks represent extracting or inserting fields into either a PyMTL `BitStruct` or Verilog `struct`.

appropriate cache bank in a four-bank configuration. For example, if bits 4 and 5 of the cache request address are zero then the cache request is sent to bank zero, and if bits 4 and 5 of the cache request are one then the cache request is sent to bank three. For a single-bank design to support its use in a banked cache organization, we need to exclude the bank bits when indexing into the cache bank. In other words, if we consider all four banks holistically as a single "banked cache", a fourth of the sets of the banked cache are in each bank. Note you cannot completely ignore the bank bits since you will need them when generating a cache line address for refills. Your design does not need to support an arbitrary number of banks only single-bank and four-bank configurations.

2. Baseline Design

The baseline design for this lab is a direct-mapped, write-back, write-allocate cache with a total capacity of 256 bytes, 16 cache lines, and 16 bytes per cache line. As with the earlier labs, we will be decomposing the baseline design into two separate modules: the datapath which has paths for moving data through various arithmetic blocks, muxes, and registers; and the control unit which is in charge of managing the movement of data through the datapath. As in the first lab, the control unit will use an FSM. Because the cache design is significantly more complicated than the first lab, we have decided to place the datapath module, control unit module, and the parent module that connects the datapath and control unit together in three different files.

The datapath for the baseline design is shown in Figure 5. The blue signals represent the control/status signals for communicating between the datapath and the control unit. Your datapath module should probably instantiate a child module for each of the blocks in the datapath diagram; in other words, you should mostly use a structural design style in the datapath. Although you are free to develop your own modules to use in the datapath, you can also use the ones provided for you in the `vc` directory. The `repl` block takes a 32b value and simply replicates it four times to create a 128b value. The `mkaddr` block simply concatenates higher order address bits plus some zeros like this: `{ addr_bits, 4'b0000 }`. As we become more proficient, we can use our judgement about what needs to be encapsulated in a child module. For example, the `repl` and `mkaddr` blocks can probably be done directly in the datapath without encapsulating them in a module. These operations are just wiring and not really "logic". However, we strongly encourage students to use comments and a temporary signal to clearly indicate where in the code you are doing this kind of wiring so that it is still easy to connect your datapath diagram to your code. Notice that to simplify our design, we are implementing the tag and data arrays using combinational-read SRAMs. This means that after setting the read address signals, the read data is available that same cycle. Note that more realistic designs meant for implementation in either an FPGA or ASIC would likely need to synchronous SRAMs.

We provide two kinds of combinational-read SRAMs in the `vc` directory: a version that supports partial byte-writes but where each entry must be an even multiple of eight bits (called `vc_CombinationalSRAM_1rw`) and a version that does not support partial byte-writes but where each entry can be any bitwidth you want (called `vc_CombinationalBitSRAM_1rw`). We recommend using a `vc_CombinationalSRAM_1rw` for your data array (since you need partial write support, and cache lines are an even multiple of eight bits), and we recommend using a `vc_CombinationalBitSRAM_1rw` for your tag array (since you do not need partial write support, and tags are not necessarily an even multiple of eight bits).

The baseline design is direct mapped with 16-byte cache lines and a total capacity of 256 bytes (i.e., 16 cache lines). So we need four bits for the byte offset and four bits for the index leaving 24 bits for the tag. Ideally, this would mean the direct mapped cache can use a an SRAM with 24 bits per entry for the tag array. Technically, for a banked-cache organization, we can use even fewer tag bits since the bank bits will always be the same in any given bank. For a four-bank cache organization, we really only need 22 bits for the tag, but it is ok to continue to use an SRAM with 24 bits per entry even in the four-bank organization. Just make sure you correctly create the address when you do a cache line eviction.

The FSM for the baseline design is shown in Figure 6. The control unit should also include additional valid and dirty bits to track the state of each tag entry. You should not use SRAMs for the valid and dirty bits. You should instead use standard register files. The various FSM states are described in more detail below:

- `STATE_IDLE (I)` – Receive the incoming cache request and place it in the input registers
- `STATE_TAG_CHECK (TC)` – Check tag, state transition based on hit/miss, valid/dirty
- `STATE_INIT_DATA_ACCESS (IN)` – Immediately write to appropriate cache line
- `STATE_READ_DATA_ACCESS (RD)` – Read from appropriate cache line
- `STATE_WRITE_DATA_ACCESS (WD)` – Write to appropriate cache line
- `STATE_EVICT_PREPARE (EP)` – Read tag and data, and prepare eviction message
- `STATE_EVICT_REQUEST (ER)` – Make a request to memory to write the evicted cache line
- `STATE_EVICT_WAIT (EW)` – Wait for memory response
- `STATE_REFILL_REQUEST (RR)` – Make a request to memory for refill the desired cache line
- `STATE_REFILL_WAIT (RW)` – Wait for memory response
- `STATE_REFILL_UPDATE (RU)` – Write the response to the victim cache line
- `STATE_WAIT (W)` – Wait while the sink (processor) is busy

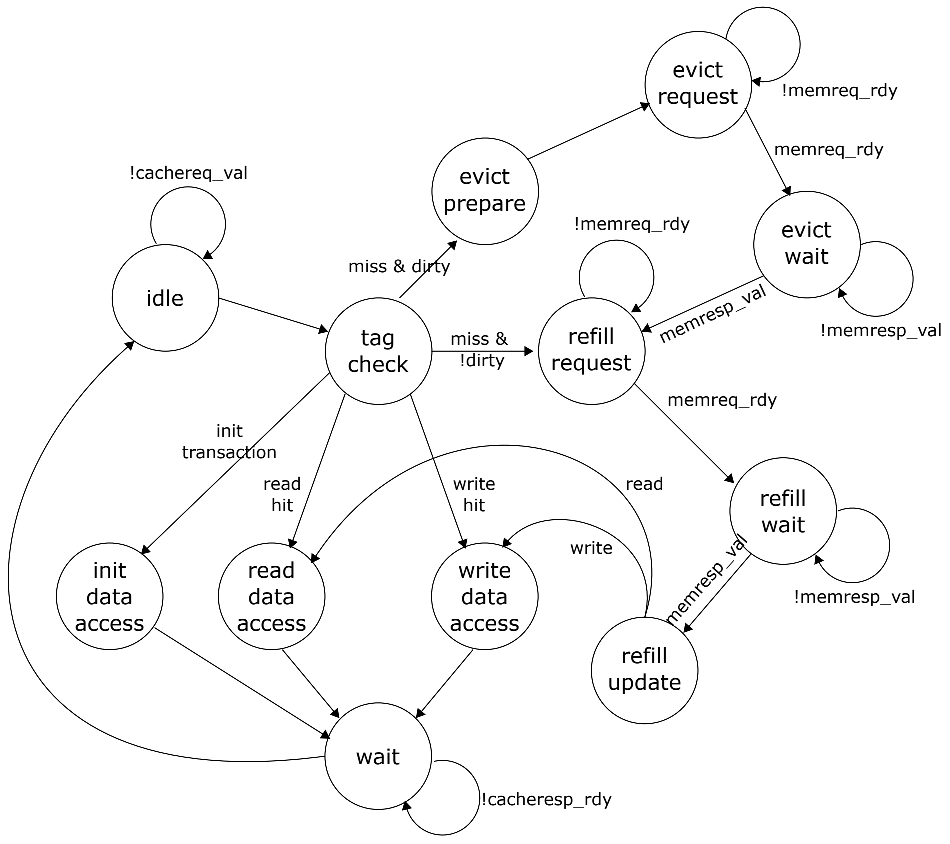

Figure 6: Baseline FSM Control Unit

This FSM control unit differs from the basic FSM cache discussed in lecture. This is due to the need to handle the various latency insensitive interfaces, the init transaction, and waiting for eviction responses. The FSM cache microarchitectures used in this lab will have a hit latency of four cycles (e.g., I $\rightarrow$ TC $\rightarrow$ RD $\rightarrow$ W). In the previous lab, the memory access time was usually a single cycle although you also tested your processor with random delays on the memory interface. Assuming we correctly implement the latency insensitive val/rdy microprotocol in both the processor and cache, then there should be no problem composing these two subsystems. Later optimizations can reduce the cache hit latency without the need to modify the processor.

We strongly encourage you to take an incremental design approach using the following steps:

- Implement the init transaction (I, TC, IN, W)
- Implement the read hit path (I, TC, RD, W)
- Implement the write hit path (I, TC, WD, W)
- Implement the refill path (I, TC, RR, RW, RU, RD/WD, W)
- Implement the evict path (I, TC, EP, ER, EW, RR, RW, RU, RD/WD, W)
- Implement support for banking

The hit path is the simplest place to start, but in order to hit in the cache, we need valid data in the cache. The init transaction (explained further in Section 4) allows you to write data into the cache without doing a refill from main memory. This kind of transaction is an example of *design-for-test* since this transaction is only meant to simplify testing and has no real use once we have a working cache model. We recommend implementing the init transaction first, and then using this transaction to implement and test both hit paths. Once the hit paths are working, then you can move onto the more complicated miss paths.

3. Alternative Design

The alternative design for this lab is a two-way set-associative, write-back, write-allocate cache with the same capacity (256 bytes) and cache line size (16 bytes) as the baseline cache. The general FSM for the alternative design will be very similar to that of the baseline design, except that the address map and control signals will likely be different. Note that you will need to split the valid bits into two parts, one for each way and carefully keep track of them. You will need to AND the result of the tag match in each way with the appropriate valid bit to determine if there is a hit or miss. The control unit should use a least-recently-used (LRU) replacement policy to choose between the two ways during eviction. You should track the LRU status with separate bits in the control unit.

4. Testing Strategy

We provide you with a basic directed tests for each of the incremental design steps described in Section 2. Work in systematic way by implementing and passing each of these groups of tests in order as you add support for specific transactions and FSM states. When finished you must add some additional directed tests for specific corner cases, and you must develop a random testing strategy.

Although you will not need to write as many tests as in the previous lab, the tests for this lab may be more challenging since you will need to carefully craft directed tests that exercise all paths in your datapath and all states and state transitions in your FSM. As with the previous labs, you will want to initially write tests using the functional-level model. Once these tests are working on the functional-level model, you can move on to testing the baseline and alternative designs.

The following commands illustrate how to run all of tests for the entire project, how to run just the tests for this lab, and how to run just the basic tests we provide on the various designs.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ..
% pytest ../lab3_mem
% pytest ../lab3_mem/test/CacheFL_test.py
% pytest ../lab3_mem/test/CacheBase_test.py
% pytest ../lab3_mem/test/CacheAlt_test.py
```

You will add your directed and random tests to `CacheFL_test.py`. Since this harness is shared across the functional-level model, the baseline design, and the alternative design you can write your tests once and reuse them to test all three models. You will be adding more test cases. Do not just make the given test cases larger. A key challenge in writing directed tests for cache memories, is that most of the miss path must be working before you can test the hit path. The miss path is significantly more complicated than the hit path, so this lends itself more towards a monolithic design process. Most of the cache must be implemented before we can run our first directed test. We could potentially use white-box ad-hoc testing that directly initializes the tag and data arrays in the cache before starting the test, but these ad-hoc tests are fragile and difficult to develop on the functional-level model.

To address this challenge, we will use a design-for-test (DFT) approach. DFT involves adding features to a design that are solely meant to facilitate test-driven development. In this specific design, we will be adding a new init transaction to go along with our current read and write transactions. The init transaction does the following:

- Writes to the appropriate cache line based on the index bits of the address
- Writes the corresponding tag
- Never updates main memory
- Sets the valid bit for that cache line to one
- Sets the dirty bit for that cache line to zero
- Sets the `cacheresp`'s test field to zero

Init transactions must be the very first transactions. Using init transactions after a read or write transaction is undefined. In the alternative design, init transactions should never be used to initialize more than one way. The staff tests will not test any undefined behavior. If you would like, you can "define" some of the undefined behavior if it facilitates your testing strategy.

Figure 7 illustrates how we will be writing tests for this lab using various helper tasks which are defined in `CacheFL_test.py`. The `req` and `resp` Python helper functions will create a memory request message and the expected memory response message. We use the test field in response messages to test whether the memory request resulted in a hit or a miss. This specific example first uses an init transaction to initialize one word in the first cache line, before using a read transaction to read this same word. Note that the second transaction should be a hit and we set the second `cacheresp` message's test field to be one. For each test case we define a Python function that returns a list of request-response message pairs. The request messages and response messages will be passed to the test source and the test sink respectively. In Figure 7, lines 6–11 and lines 17–22 illustrate two test cases. We also provide you a helper function to load data into the test memory before running the test. It will allow you to test the cache refill path without involving the cache evict path. If you want to load data into the test memory before running the test, you can create a Python function that returns a list of address-data pairs and then the test function we provide will load it to the test memory before running the test. For example lines 26–31, load the test memory with two words of data at addresses `0x00000000` and `0x00000004` before running the `read_miss_1word` test case.

Once you create a new test harness, you can add it to a test case table, as shown on lines 37–41 in Figure 7. A test case table has seven columns. The first column is the name of tests, the second one is the function that generates source/sink messages, and the third one is the function that generates memory data to be loaded before running the test. If your test case does not need to load data to the test memory, simply use `None`. The last four columns are for the memory's stall probability, memory latency, source delay, and sink delay, respectively.

Ideally, we want to use same set of tests to test the FL, baseline RTL, and alternative RTL model. We define all the test cases in `CacheFL_test.py`, test them on the FL model to make sure the test cases are correct, and we import them in the `CacheBase_test.py` and `CacheAlt_test.py` test scripts. We need to be careful about the test field, because the same test may have different hit/miss behavior on different cache implementations. For example, in an FL cache, every response is a miss because we simply pass `cachereq` and `cacheresp` to memory. The alternative design will have less conflict misses than the baseline design, because the alternative design is set-associative whereas the baseline design is direct-mapped. Therefore, we should expect different values for the test field in the `cacheresp` messages passed to the sink from different cache models even if we use the same source messages. To solve this issue, we can pass a comparison function into the stream sink which is used to compare the expected and actual messages. In `harness.py`, we provide a 'cmp_wo_test_field' function that compares all fields in a memory response message *except* the test field. In other words,

```
1  #----------------------------------------------------------------------
2  # Test Case: read hit path
3  #----------------------------------------------------------------------
4  # The test field in the response message: 0 == MISS, 1 == HIT
5
6  def read_hit_1word():
7    return [
8      #    type  opq   addr       len data                type  opq test len data
9      req( 'in', 0x00, base_addr,  0, 0xdeadbeef ), resp( 'in', 0x00, 0, 0, 0          ),
10     req( 'rd', 0x01, base_addr,  0, 0          ), resp( 'rd', 0x01, 1, 0, 0xdeadbeef ),
11   ]
12
13 #-------------------------------------------------------------------------
14 # Test Case: read miss path
15 #-------------------------------------------------------------------------
16
17 def read_miss_1word():
18   return [
19     #    type  opq   addr       len data                type  opq test len data
20     req( 'rd', 0x00, 0x00000000, 0, 0          ), resp( 'rd', 0x00, 0, 0, 0xdeadbeef ),
21     req( 'rd', 0x01, 0x00000004, 0, 0          ), resp( 'rd', 0x01, 1, 0, 0x00c0ffee ),
22   ]
23
24 # Data to be loaded into memory before running the test
25
26 def read_miss_1word_mem():
27   return [
28     # addr      data
29     0x00000000, 0xdeadbeef,
30     0x00000004, 0x00c0ffee,
31   ]
32
33 #-------------------------------------------------------------------------
34 # Test table for generic test
35 #-------------------------------------------------------------------------
36
37 test_case_table_generic = mk_test_case_table([
38   (                       "msg_func         mem_data_func        stall lat src sink"),
39   [ "read_hit_1word",      read_hit_1word,  None,                0.0,  0,  0,  0    ],
40   [ "read_miss_1word_mem", read_miss_1word, read_miss_1word_mem, 0.0,  0,  0,  0    ],
41 ])
42
43 @pytest.mark.parametrize( **test_case_table_generic )
44 def test_generic( test_params, cmdline_opts ):
45   run_test( CacheFL(), test_params, cmdline_opts, cmp_wo_test_field )
```

Figure 7: Writing Directed Tests – Example of how to write tests in the `CacheFL_test.py` file. We create all test cases in this file and use it to test both FL, baseline RTL and alternative RTL model.

if we use this comparison function we are effectively ignoring the test field. If we do not specify a comparison function, then the stream sink will check the test field.

As mentioned above, your baseline and alternative designs will have different hit/miss behaviors, so you may need to use different expected response messages for the same sequence of source messages. You should add tests designed specifically for your baseline or alternative design. For example, you should have tests that only hit in a two-way set-associative cache to make sure your alternative design is indeed two-way set-associative. We group the test cases into five test case tables. `test_case_table_generic` is shown on lines 37–41 in Figure 7 and is used to generically test both the baseline and alternative designs (i.e., tests in this table should have the same expected behavior for both the baseline and alternative design). `test_case_table_random` is for random testing. `test_case_table_dmap` and `test_case_table_sassoc` are used to test only one of the designs. `test_case_table_bank` should explicitly test a four-bank configuration. We provide examples for most of these kinds of tests.

Make sure the combination of the given tests and your new tests cover all of these scenarios:

- Read hit path for clean lines
- Write hit path for clean lines
- Read hit path for dirty lines
- Write hit path for dirty lines
- Read miss with refill and no eviction
- Write miss with refill and no eviction
- Read miss with refill and eviction
- Write miss with refill and eviction
- Tests which stress entire cache, not just a few cache lines
- Conflict misses
- Capacity misses
- LRU replacement policy by filling up a way
- Tests for a four-bank cache organization
- Tests specifically designed to trigger corner cases in your alternative design
- Testing all or some of the above using random source and sink delays and test memory delays

Once you have finished writing your directed tests you should move on to writing random tests. You can use the same Python-based random test generation system we used in the first lab. Some suggestions for what you might want to test are listed below. Each of these would probably be a separate test pattern, or potentially multiple test patterns with different random parameters. You don't necessarily need to do all of these.

- Simple address patterns, single request type, with random data
- Simple address patterns, with random request types and data
- Random address patterns, request types, and data
- Unit stride with random data
- Stride with random data
- Unit stride (high spatial locality) mixed with shared (high temporal locality)

Writing random tests for memories can actually be quite challenging. With the first lab, the correct output was trivial to calculate based on the random inputs, but with a memory system the correct output (i.e., the data we expect in a memory read response) depends on the last write to the corresponding address. To write random tests with random address patterns and/or types, you will need to keep track of a "reference memory" in your Python script. This reference memory can just be an array of words. Every time you generate a write request, you should update the reference memory in addition to generating the appropriate write request. Every time you generate a read request, you should consult your reference memory to determine what data we expect to be returned in a memory read response.

You will almost certainly want to use line tracing to visualize the execution of transactions on your baseline and alternative designs. We have provided some line tracing code for you in the test harness which traces the cache request/response and memory request/response interfaces. Figure 8 illustrates a line trace for the basic test in Figure 7 executing on the baseline design with extra annotations to indicate what the columns mean. The first columns shows when memory request messages are sent from the test source into the cache, and the last column shows when memory response messages are sent from the cache back to the test sink. The second column shows the state of the cache. This column is critical to understanding the behavior of your cache, but it is not currently implemented

```
cycle cachreq               state memreq                 memresp                  cacheresp
  2:                         (I )                       ().                      .
  3: in:00:00001000:deadbeef (I )                       ().
  4: #                       (TC)                       ().
  5: #                       (IN)                       ().
  6: #                       (W )                       ().                      in:00:0:
  7: rd:01:00001000:         (I )                       ().
  8: .                       (TC)                       ().
  9: .                       (RD)                       ().
 10: .                       (W )                       ().                      rd:01:1:deadbeef
```

Figure 8: Line Trace for Basic Directed Test – The line trace shows two memory requests sent from the test source to the cache, the four states each transaction goes through, and then the response being sent from the cache back to the test sink.

```
cycle cachreq               state memreq                        memresp          cacheresp
  2:                         (I )                             ().                .
  3: rd:00:00000010:         (I )                             ().
  4: #                       (TC)                             ().
  5: #                       (RR) rd:00:00000010:             ().
  6: #                       (RW)                             ()rd:00:0:00...00
  7: #                       (RU)                             ().
  8: #                       (RD)                             ().
  9: #                       (W )                             ().                rd:00:0:00000000
 10: wr:01:00000010:deadbeef (I )                             ().
 11: #                       (TC)                             ().
 12: #                       (WD)                             ().
 13: #                       (W )                             ().                wr:01:1:
 14: rd:02:00000110:         (I )                             ().
 15: .                       (TC)                             ().
 16: .                       (EP)                             ().
 17: .                       (ER) wr:00:00000010:00...deadbeef().
 18: .                       (EW)                             ()wr:00:0:
 19: .                       (RR) rd:00:00000110:             ().
 20: .                       (RW)                             ()rd:00:0:00...00
 21: .                       (RU)                             ().
 22: .                       (RD)                             ().
 23: .                       (W )                             ().                rd:02:0:cafecafe
```

Figure 9: Line Trace for More Involved Directed Test – The line trace shows three memory requests meant to trigger an eviction of a dirty line. Notice how the third request must go through a total of 10 states as the FSM does tag check, eviction, and refill. Note, only a portion of the bits for the data field for memory request/responses are shown for simplicity.

in the lab harness. You will need to modify the line tracing code in your baseline and alternative designs to append a string representing the current cache state. Use the short state names as given in the state description list above (e.g., `I` for `STATE_IDLE`, `TC` for `STATE_TAG_CHECK`). The third and fourth columns show the memory request and response messages to/from the test memory. Figure 9 illustrates a line trace for a more extensive test that is forcing a line to be evicted. Notice how the line trace clearly shows what data is moving between the test source/sink, cache, and test memory. We also recommend including all 16 tags and even a character to indicate which cache lines are dirty in the line trace.

In addition to the tests for the entire cache, you must also add additional unit tests for any datapath components you add or modify.

```
// loop1 pattern
// a array allocated at 0x1000
for ( i = 0; i < 100; i++ )
  result += a[i];
```

```
// loop2 pattern
// a array allocated at 0x1000
// b array allocated at 0x2000
for ( i = 0; i < 100; i ++ )
  result += a[i]*b[i]
```

```
// loop3 pattern
// a array allocated at 0x1000
// b array allocated at 0x2000
// c array allocated at 0x3800
for ( i = 0; i < 100; i ++ )
  result += a[i]*b[i] + c[i]
```

Figure 10: Evaluation Patterns – Three loops that correspond to the given evaluation patterns.

5. Evaluation

Once you have verified the functionality of the baseline and alternative designs, you should then use the provided simulator to evaluate these two designs. The simulator delays all responses from the test memory by 20 cycles to model a long main-memory latency. You can run the simulator to see the performance of each cache implementation as follows:

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% ../lab3_mem/mem-sim --impl base --input loop1 --stats
% ../lab3_mem/mem-sim --impl alt  --input loop1 --stats
```

We provide you with three input patterns that capture common access patterns in loops. The C code for each loop that might generate the corresponding pattern is shown in Figure 10. The `loop1` pattern simply iterates through an array. The `loop2` pattern iterates through two arrays, and the `loop3` pattern iterates through three arrays. All three patterns start the array accesses at carefully chosen memory addresses to demonstrate interesting cache behaviors.

The simulator will display a collection of statistics: number of cycles, number of memory and cache accesses, number of misses, miss rate, and the average memory access latency (AMAL). You should study the line traces (with the `--trace` option) and possibly the waveforms (with the `--dump-vcd` option) to understand the reason why each design performs as it does on the various patterns.

You must add additional evaluation patterns with different amounts of spatial and temporal locality. We recommend including some patterns that mix reads/writes and random patterns. We recommend adding three more patterns for evaluation. Obviously, these patterns need to be carefully chosen to highlight the differences between the baseline and alternative designs. You will also need to add the name of the new dataset to the `mem-sim` simulator script.

Writing an access pattern for the simulator is similar to writing a test case. Basically for each pattern you need to write a Python function that returns a list of source messages, a list of sink message, and a list of memory address-data pairs that will be loaded to the test memory before the simulation. Please keep in mind that patterns you will write in the simulator are not for testing. Instead, you need to fully test your designs using test cases and then use patterns in the simulator script as a way to evaluate your designs.

6. Looking Towards the Multicore System in Lab 4

In lab 4, we will compose the pipelined processor designed in lab 2 (which includes the multiplier designed in lab 1) and the cache memory designed in lab 3 to create a baseline single-core system, and we will compose the pipelined processor, cache memory, and a simple ring networks network designed in lab 4 to create the alternative multicore system. You will be developing a serial and parallel sorting microbenchmark, and comparing the performance of this microbenchmark across the baseline and alternative designs. You will quickly find that the performnace of your alternative multicore system is limited by the hit latency of the cache you designed in this lab. While we could move to a more aggressive pipelined cache microarchitecture, you can achieve much of the same benefit by simply merging states in the FSM control unit used in this lab. Ideally, you would merge enough states to enable a single-cycle hit latency for reads (i.e., a single state for read hits) and a sustained throughput of one read hit per cycle. This requires performing tag check and data access in parallel, and carefully handling the val/rdy signals for the cache request and response interfaces. Writes can potentially use two states to do tag check and data access in sequence, although single-cycle hit latency for writes is still possible if the cache response is sent back in the first state. Reducing the read hit latency is the most critical since this would improve the performance of instruction fetch in your processor. There is no need to wait until lab 4. Students should feel free to start optimizing their cache as part of the alternative design in this lab, or after this lab is submitted.

Acknowledgments

This lab was created by Shunning Jiang, Shuang Chen, Ian Thompson, Moyang Wang, Christopher Torng, Berkin Ilbeyi, Ackerley Tng, Shreesha Srinath, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.