ECE 4750 Computer Architecture, Fall 2013

Lab 3: Cache Microarchitecture

School of Electrical and Computer Engineering
Cornell University

revision: 2013-10-18-20-54

In this lab, you will design two cache microarchitectures, which have a finite-state-machine (FSM) based controller. The baseline design is a direct-mapped writeback cache which has a total capacity of 256 bytes with 16 cache lines and with 4 words per cache line. The alternative design is a two-way set associative cache which has the same total memory capacity and same cache line bitwidth. You will be given the state machine and datapath for the baseline design. You are required to design the datapath and control for the alternative design. A testing framework is provided to test both the cache microarchitectures and you are required to add to the tests by creating an effective testing strategy for both of the designs. Finally, we provide you a simulator to test the performance of the cache microarchitectures for various access patterns.

As with all lab assignments, the majority of your grade will be determined by the lab report. You should consult the course lab assignment assessment rubric 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:

- complex finite-state-machine based cache controllers
- analyze tradeoffs involved in cache microarchitecture designs
- effective design patterns including control/datapath split, FSM control, and message interfaces
- effective design methodologies including test-driven development and incremental development

This handout assumes that you have read and understand the course tutorials. To get started, you should access a course development machine and perform the following steps:

```
% cd ${HOME}
% source setup-ece4750.sh
% ece4750-lab-admin start ece4750-lab3-mem
% mkdir ${HOME}/ece4750/ece4750-lab3-mem/build
% cd ${HOME}/ece4750/ece4750-lab3-mem/build
% ../configure
% make
% make check
```

For this lab you will be working in the `plab3-mem` subproject which includes the following files:

- Null cache. Forwards the cache requests to memory requests
  - `plab3-mem-BlockingCacheSimpleDpath.v` – Datapath
  - `plab3-mem-BlockingCacheSimpleCtrl.v` – Control
  - `plab3-mem-BlockingCacheSimple.t.v` – Test suite
  - `plab3-mem-BlockingCacheSimple.v` – Cache implementation
  - `plab3-mem-sim-null.v` – Simulator
- Blocking direct-mapped write-back, write-allocate cache
  - `plab3-mem-BlockingCacheBaseDpath.v` – Datapath

    - `plab3-mem-BlockingCacheBaseCtrl.v` – Control
    - `plab3-mem-BlockingCacheBase.t.v` – Test suite
    - `plab3-mem-BlockingCacheBase.v` – Cache implementation
    - `plab3-mem-sim-dmapped.v` – Simulator
- Blocking 2-way set-associative write-back, write-allocate cache
    - `plab3-mem-BlockingCacheAltDpath.v` – Datapath
    - `plab3-mem-BlockingCacheAltCtrl.v` – Control
    - `plab3-mem-BlockingCacheAlt.t.v` – Test suite
    - `plab3-mem-BlockingCacheAlt.v` – Cache implementation
    - `plab3-mem-sim-setassoc.v` – Simulator
- `plab3-mem-test-harness.v` – Test harness shared across all implementations
- `plab3-mem-sim-harness.v` – Simulation harness shared across all implementations
- `plab3-mem-input-gen.py` – Python script to generate memory patterns

You can use the `ece4750-lab-admin` script and the `git` command to add members to your lab group, use your own group repository to collaborate with your group members, and ultimately prepare your submission for uploading to CMS. **Remember to use the ece4750-lab-admin script to create the submission tarball.**

1. Introduction

Caches are intermediate memory structures which exist between processors and the main memory and operate on the principle of locality. Caches improve the performance of the processor by allowing us to exploit the spatial and temporal locality of memory accesses. As you have learned in class, caches have limited capacity and various choices have to be considered to design cache microarchitectures.

In this lab you will explore a finite-state-machine based cache controller to design direct-mapped and 2-way set-associative caches with write-allocate and write-back policies. The instruction fetch access and data memory access is usually assumed to be one cycle or in other words we assume a cache hit latency of one cycle in a standard processor pipeline design. Based on the simple FSM controller, the caches you will implement will seemingly reduce the performance of the processor you made in lab 2 because the hit latency will be greater than 1 without any optimizations. Keep in mind that the relative performance reduction is due to the fact that we were using an ideal test memory with a single cycle hit latency in lab 2. If a more realistic memory were used, adding a cache between your processor and the memory will definitely improve performance.

This lab will have extensions where you can make various improvements to the cache.

As you have learnt in class, a modern computer system usually has multiple levels of a cache hierarchy to optimize the amount of time required to transfer data between the main memory and the processor. For the purposes of this lab, we will consider to build simple level-1 (L1) cache microarchitectures. Modern chips rely on synchronous SRAMs technology to design caches that provide data at the speeds desired. Design tradeoffs in choosing a memory technology include: capacity, latency, bandwidth (number of width of ports) and cost. In standard ASIC methodology, chips use special memory compilers to generate the SRAM based cache memories.

Here are some verilog components in `vc/` that you want to use:

- `vc_CombinationalSRAM_1rw` – SRAMs that do combinational reads, synchronous writes
- `vc_ResetRegfile_1r1w` – Register files, with reset capability
- Pack and Unpack modules for memory requests - to standardize format of memory messages
    - `vc_MemReqMsgPack`

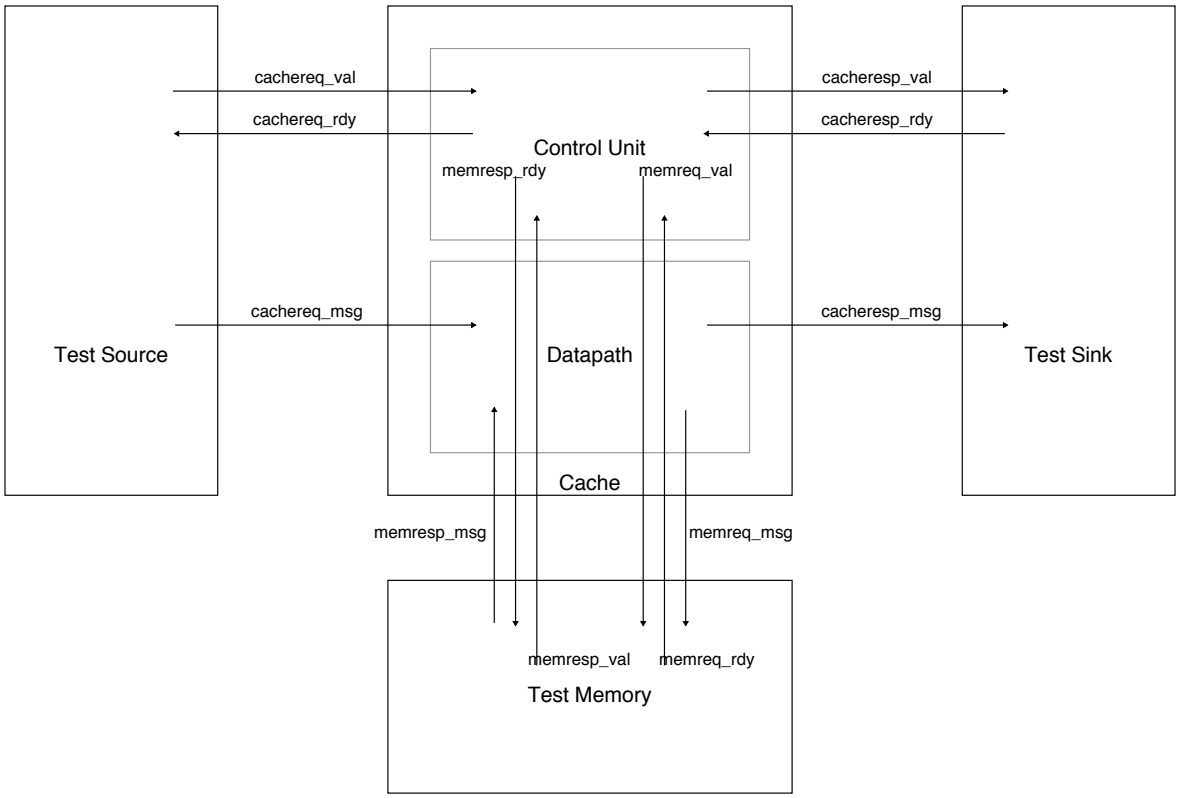

Figure 1: Baseline design : Direct-Mapped Write-Back Cache Test Harness

- `vc_MemReqMsgUnpack`
- `vc_MemRespMsgPack`
- `vc_MemRespMsgUnpack`

You are of course free to use any of the other verilog components like

- `vc_EnResetReg`
- `vc_EqComparator`
- `vc_Mux2`
- `vc_Mux4`

Note that the baseline and alternative cache designs you will be implementing will use the combinational SRAM models. Make sure you know what that means by looking at the implementation and corresponding unit tests.

2. Baseline Design

The goal of the baseline design is to implement a direct-mapped write-back cache with 16 lines, each with 4-words (128-bits).

We have provided you with files as described above, although all of them will essentially be copies of `plab3-mem-BlockingCacheSimple.v`, which is the null cache. You are given the datapath and finite state machine diagram to implement the baseline design.

Recall that *direct-mapped* means that each memory block can only be assigned to exactly one cache line location. A *write-back* cache only needs to modify a valid local copy of data on a write. When modifying the local state, it marks that cache line "dirty". When there is another miss that forces a dirty cache line out, only then would it need to do an evict and a refill. An evict means writing the modified "dirty" data back to the main memory. Refill means reading the clean, unmodified state of the cache line from the memory and writing that value to the appropriate cache line. Compared to *write-through* caches which write the requested data to the main memory regardless of a hit or a miss in the cache, *write-back* caches only write the modified state if the cache line is dirty and is forced an eviction.

The test harness that is provided for you is as shown in Figure 1. The cache has two distinct interfaces, one being processor-cache and the other being the cache-memory interface. It is easy to confuse the following:

- cachereq – the request from the source (processor) to the cache
- cacheresp – the response from the cache to the sink (processor)
- memreq – the request from the cache to the memory
- memresp – the response from the memory to the cache

The control unit for the baseline design should use the FSM as shown in Figure 2. The control unit also includes additional state bits to track the state of each tag entry - valid and dirty bits. The valid and dirty bits are modeled using register files, as mentioned in class.

Here's a description of the various states in the FSM and their acronyms:

- `I` – `STATE_IDLE` – Consumes the incoming cache request and places it in the input registers
- `TC` – `STATE_TAG_CHECK` – Checks the tag, transits to the correct state based on whether it was a hit/miss, and if the cacheline is valid or dirty
- `IN` – `STATE_INIT_DATA_ACCESS` – Handles writing to the appropriate cacheline when the init transaction is received
- `RD` – `STATE_READ_DATA_ACCESS` – Handles reading from the cacheline on a read hit
- `WD` – `STATE_WRITE_DATA_ACCESS` – Handles writing to the cache on a hit
- `RR` – `STATE_REFILL_REQUEST` – Makes a request to memory for the appropriate data
- `RW` – `STATE_REFILL_WAIT` – Waits while the memory is busy, and once the response is valid, this state registers the response
- `RU` – `STATE_REFILL_UPDATE` – Writes the response to the cacheline
- `EP` – `STATE_EVICT_PREPARE` – Reads the tag and data, and prepares the message to be sent to memory
- `ER` – `STATE_EVICT_REQUEST` – Makes a request to memory to write the data prepared
- `EW` – `STATE_EVICT_WAIT` – Waits while the memory is busy, and once valid, transits to `STATE_REFILL_WAIT`
- `W` – `STATE_WAIT` – Waits while the sink (processor) is busy

The datapath for the baseline design is as shown in Figure 4. The blue signals represent the control/status signals for communicating between the datapath and the control unit. The datapath should be written structurally. The section marked "replicate" replicates the input data 4 times, to produce a 128 bit output.

We highly recommend that you take an incremental design approach. We have provided you with tests that facilitate this incremental design approach as described in Section 4. Here is a suggestion of an incremental design approach:

1. Build support for the init transaction (I, TC, IN, W)
2. Build the read hit path (I, TC, RD, W)

3. Build the write hit path (I, TC, WD, W)
4. Build the refill path (I, TC, RR, RW, RU, RD/WD, W)
5. Build the evict path (I, TC, EP, ER, EW, RR, RW, RU, RD/WD, W)

In general you first would want to tackle the hit path because it is simpler. However, in order to be able to hit in the cache, we suggest you implement the init transaction which is explained further in Section 4. This transaction allows you to load a value to the cache without doing the refill from the memory, which is more complicated to implement. Once you have got both hit paths working, then you can work on the miss paths.

The baseline design section in your lab report will need to briefly summarize the baseline design and any changes you made to the provided design.

3. Alternative Design

The goal of the alternative design is to implement a two-way set associative cache of the same capacity. The FSM for the alternative design would be the same, however your control will differ slightly. The control unit should use a least-recently-used (LRU) replacement policy to choose between the two sets for new allocations. You can track this by using separate bits in the control unit. As with the previous lab, once you get your baseline design working, you could copy and rename the baseline design as the alternative. You can do the changes to implement set associativity on the direct-mapped cache. The lab report you turn in must provide a detailed datapath diagram, finite-state-machine modifications (if any) and sufficient detail to accurately describe your design.

4. Testing Strategy

The test framework for the cache designs uses the familiar test sources and test sinks. Figure 1 shows how it is set up.

We have provided you with some tests for the baseline design. These tests test for various incremental design points (hit path only, refill etc.). In addition, we have randomized testing with and without random delays to fully test your cache's functionality. We also use the evaluation patterns as test cases to make sure your cache still functions in more challenging and longer patterns.

4.1. Design for Test

Design-for-test is a design technique that adds features to a product that allow the product to be easily tested. In this lab, we designed the init transaction, purely for testing purposes.

The init transaction does the following:

- Writes (overwrites) the appropriate cacheline based on the index bits of the address - it does not write to memory
- Sets the valid bit for that cacheline to 1
- Sets the dirty bit for that cacheline to 0

Its behavior, if requested to write to a dirty cacheline, is **undefined**. Essentially, the init transaction is a special one that loads some specified data into the cache.

An init transaction would have no place in a real cache, so what is its purpose? The init transaction allows us to test the cache without having to poke into the cache to directly write cachelines in the early stages of building cache functionality. An alternative would be to build some kind of

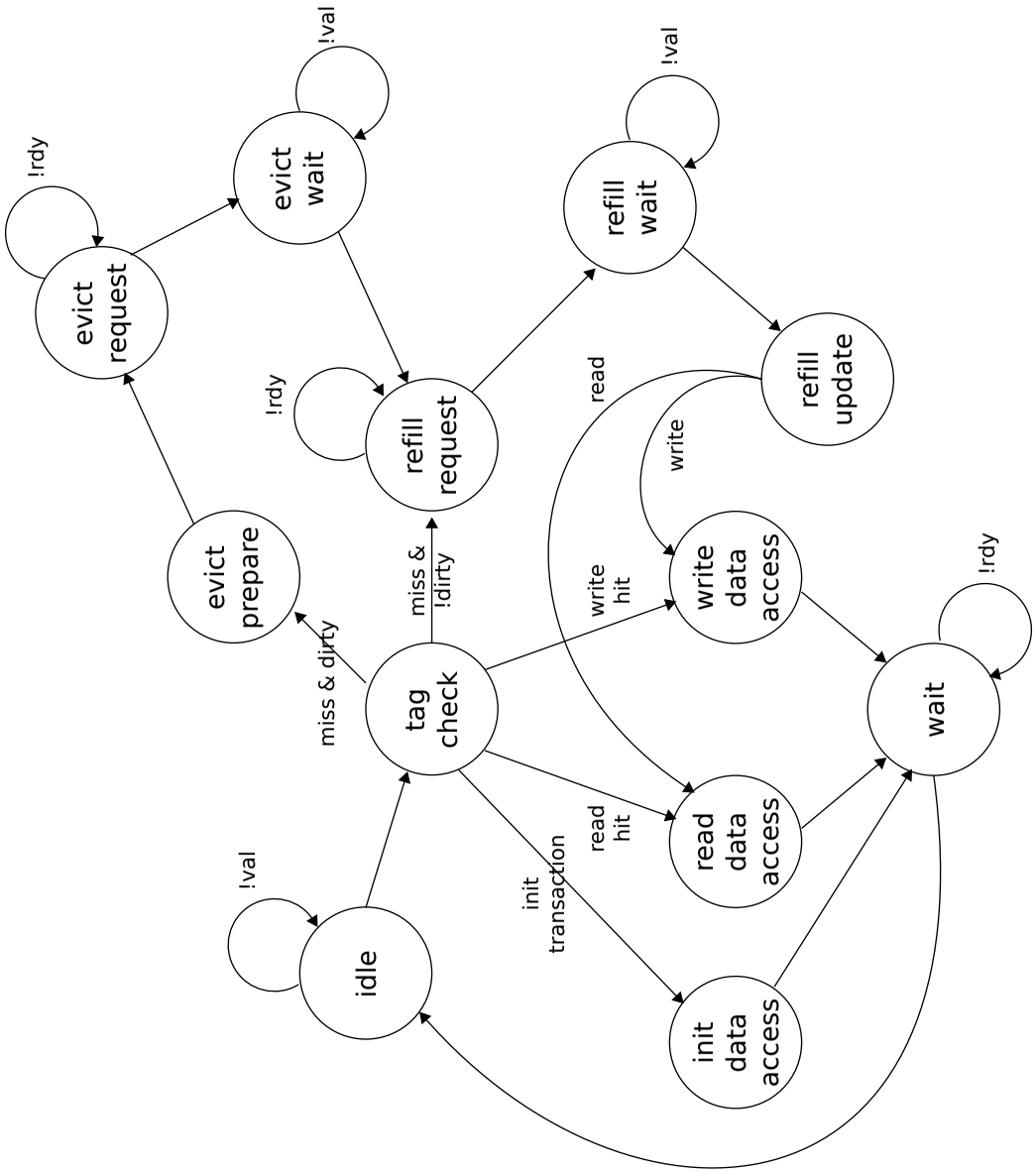

Figure 2: Baseline design FSM control

memory image and image that to the cache to initialize it before testing the read transaction, but that would require an extremely white-box understanding of the cache implementation, right down to the structure and names of components.

Any tests written with such in depth knowledge of the structure would be overly specific. If the cache implementation changes, we would have to throw away those tests, wasting all the effort put into writing them.

Having an init transaction supported by all cache implementations would allow us to re-use old tests in new implementations, only adding to the growing base of test cases and having a more robust testing framework for future products.

```
//------------------------------------------------------------------------
// Ad-hoc test: init loads one byte
//------------------------------------------------------------------------

localparam c_req_wn  = `VC_MEM_REQ_MSG_TYPE_WRITE_INIT;

`VC_TEST_CASE_BEGIN( 1, "init - loading one word in the 0th cacheline" )
begin
  init_test_case( 0, 0, 0 );

  // Initialize Port

  //         ----------------- memory request ----------------
  //         type      opaque addr          len   data
  init_port( c_req_wn, 8'h00, 32'h00000000, 2'd0, 32'he110341d,

  //                              --------- memory response ----------
  //                              type       opaque len   data
                                 c_resp_wn, 8'h00, 2'd0, 32'h???????? );

  run_test;

  if ( th.done )
    `VC_ASSERT( th.cache.dpath.data_array.mem[0][31:0] == 32'he110341d );

end
`VC_TEST_CASE_END
```

Figure 3: Sample ad-hoc test case to test the init transaction

4.2. Ad-hoc Testing

All transactions must be tested, and the init transaction is no exception. One way we could test the init transaction would be to work on both the init transaction and the read transaction, and hope that they both work, and can be used to test each other. If the tests pass, how do we know those are not false positives? If the tests do not pass, how do we know which transaction was implemented wrongly?

Ad-hoc testing is the solution to this. There are a few ways to do ad-hoc testing, for this lab, to make sure the init transaction is working. The first of which could be to use gtkwave to check that the

- `write_data` port in the data array gets the right data
- `write_byte_en` port has the correct bytes enabled
- `write_en` port is enabled at the right time
- `write_addr` port has the right index fed to it

That is one option that has rather many things to look out for.

Another way to be to write test cases, as shown in Figure 3. The transaction set up in `init_port` writes data to the 0th cacheline in the cache, which is then checked by `` `VC_ASSERT ``. This is a convenient way that allows you to make changes, make and run test and quickly see if your code changes took effect.

We recommend that you get the init transaction working using ad-hoc tests, and then throw those ad-hoc tests away, migrating to using the init transaction to test the other paths.

4.3. Requirements

Note that the init transaction is only there to help you and facilitate incremental design. The test cases we provide you use the init transaction, so you do need to implement it. However, the semantics of the init transaction is identical as a write to the testing framework. So you could just treat the `WRITE_INIT` message type the same as a `WRITE` message type and the tests will pass. However, you are highly recommended to utilize this special message type to test part of your state machine.

You are responsible for writing tests for the alternative design. Make sure that you include tests which trigger and robustly test the associativity of the cache design.

As outlined in the course lab assignment assessment rubric, the testing strategy section in your lab report should briefly summarize the approach used in testing the baseline design before focusing in detail on the tests you used to verify your alternative design.

5. Evaluation

You will evaluate the different cache designs using different memory patterns. You can run the simulator to see the performance of the cache implementation like this:

```
% cd ${HOME}/ece4750/ece4750-lab3-mem/build
% make
% ./plab3-mem-sim-null     +input=random +stats
% ./plab3-mem-sim-dmapped  +input=random +stats
% ./plab3-mem-sim-setassoc +input=random +stats
```

The following are the patterns we provide that you can specify using `+input` flag:

- `random` – random accesses in a memory space 8 times larger than the cache size
- `ustride` – unit stride pattern
- `stride2` – stride pattern of 2 (accesses every other word)
- `stride4` – stride pattern (accesses one word in four)
- `shared` – shared access pattern to two memory addresses
- `ustride-shared` – shared access pattern interleaved with unit stride
- `loop-2d` – 2D loop pattern
- `loop-3d` – 3D loop pattern

The `random` pattern does 256 memory accesses in a memory space that is 8 times larger than the cache. We do not allow full address space for the random accesses to make it more realistic that the random accesses can occasionally hit in the cache as well.

The stride patterns walk through the memory addresses one by one. The unit stride accesses every consecutive word, `stride2` accesses every other consecutive word etc. In general, we can represent the stride pattens as the accesses to the a array in the following C sequence:

```
int *a;
for ( i = 0; i < n * stride; i += stride )
  sum += a[i];
```

The `shared` pattern accesses a few non-conflicting addresses over and over again. The pattern looks like the following:

```
int *a;
```

```
for ( i = 0; i < n; i++ )
  // assume addr0 and addr1 are not conflicting
  sum += a[addr0] + a[addr1];
```

The `ustride-shared` interleaves the `shared` pattern with the `ustride`. It looks like the following:

```
int *a;
for ( i = 0; i < n; i++ )
  // assume addr0 and addr1 are not conflicting
  sum += a[i] + a[addr0] + a[addr1];
```

The patterns until here are crafted so that they all have different amounts of spatial and temporal locality. In your report, be sure to identify what sort of locality each one has and argue how one cache architecture is better than another because of different localities or lack thereof.

The remaining two patterns, `loop-2d` and `loop-3d` are crafted to show the benefit of direct-mapped over the set-associative and vice versa. You should study these patterns and explain how they demonstrate how set-associativity can sometimes help and other times hurt the performance. The two snippets illustrate `loop-2d` and `loop-3d` respectively.

```
int *a;
for ( i = 0; i < 5; i++ )
  for ( j = 0; j < 100; j++ )
    sum += a[j];
```

```
int *a;
for ( i = 0; i < 5; i++ )
  for ( j = 0; j < 2; j++ )
    for ( k = 0; k < 8; k++ )
      sum += a[j*64 + k*4];
```

Note that even though the C snippets only show a read to an array element, in the simulations, 10% of the cache accesses are writes. You should explain how the request type would affect the performance.

The simulator will generate a collection of stats including number of cycles, number of hits and misses, number of memory and cache accesses, number of evicts and refill, and the average memory access latency (AMAL). You can also use the following command to generate AMAL for all of the patterns on all of the implementations:

```
% make eval-plab3-mem
```

In your report, you can represent the performance of different implementations in a table format. You might also want to create a bar plot of the AMAL for each implementation and pattern. You must present a strong argument for why each design behaves as observed for the patterns provided. Does the alternative design always perform better than the baseline design? Why would you pick one over the other considering a generalized set of real-world programs? Make a compelling case for any of the designs.

6. Extensions

We will be accepting extensions for this lab. The following are some suggestions for extensions.

6.1. Synchronous SRAMS

In the baseline and alternative design, we use Combinational SRAMs, but in industry, synchronous SRAMs are more common. Having the clock edge for reads enables higher bandwidth operation. To use synchronous SRAMs in your design, use `vc_SynchronousSRAM_1rw` instead of `vc_CombinationalSRAM_1rw` for the tag and data arrays.

6.2. Optimize FSM to collapse states

The FSM in your baseline and alternative design has way too many states. Many of them do rather similar things, and could be collapsed together. Find those, collapse states, and improve the performance of your cache!

6.3. Parameterized caches

In your baseline and alternative designs, you can assume fixed cache sizes (256 bytes). Extend your design by added parameters to all your components such that someone wanting to instantiate a cache of a different size can do so just by specifying a different parameter.

6.4. Pipe and bypass queues

The read hit path is **4 cycles**, which is definitely too many. We can cut this number down to 1 cycle if we use a pipe queue on the cache request interface to eliminate the idle state and a bypass queue on the cache response interface to eliminate the wait state, amongst other improvements.

6.5. Subword accesses

In the baseline and alternative design, we assume that the least significant two bits are always 0 - we can't access halfwords or individual bytes. Support subword accesses in your extension to be able to read just one byte or halfwords, in addition to full-word reads and writes.

6.6. Atomic memory operations

As briefly mentioned in class, Atomic Memory Operations or AMOs are uninterruptible read-modify-write operations that are special transactions.

You are welcome to come up with your own ideas regarding extensions. Please note that extensions are only meant for extra credit for students who want to challenge themselves and not required to get an A+ from the lab assignment. You should make sure both of your designs and the lab report are in a very good shape before you attempt extensions.

Acknowledgments

This lab was created by Berkin Ilbeyi, Ackerley Tng, Shreesha Srinath, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.

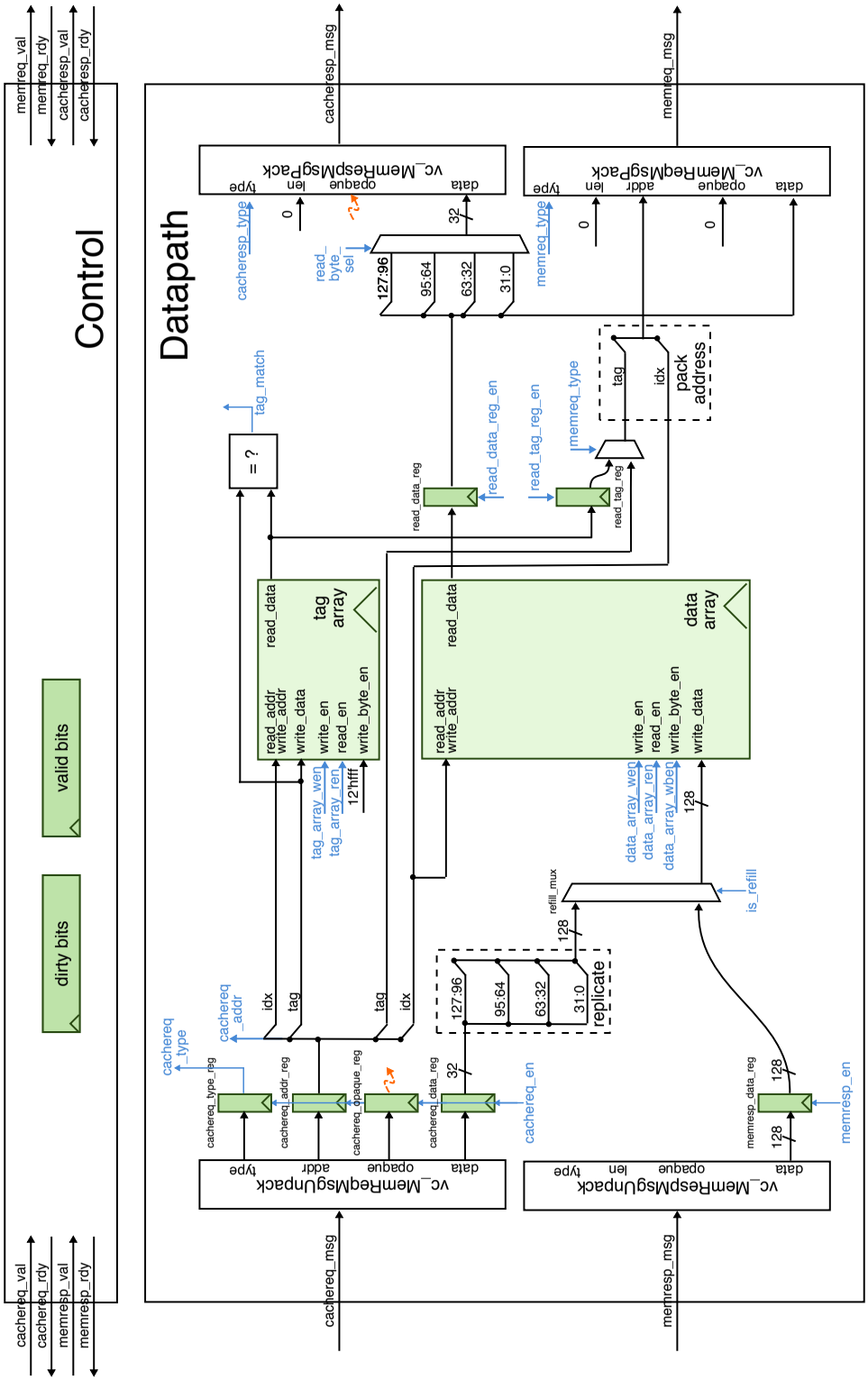

Figure 4: Baseline design : Direct-Mapped Write-Back Cache