**ECE 4750 Lab 3: Blocking Cache**

Carly Swetz (cms464), Raghav Kumar (rk524), Kyle Betts (kcb82)

## 1. Introduction

This lab was designed to give us a better understanding of various cache designs, namely, direct-mapped and two-way set-associative, in order to complement what we have been learning in lectures. We successfully implemented a write-back, write-allocate direct-mapped cache with 16 byte cache lines and a total capacity of 256 bytes along with a two-way set-associative cache with the same total size. This lab was essential from an overall, birds eye view of the class because implementing these two cache designs added to the overall functionality and tied in neatly with our previous two lab assignments. While the Lab 1 multiplier fit into the pipelined processor we developed in Lab 2, our cache implementations from Lab 3 would greatly improve the average memory latency (AMAL) we achieved in Lab 2. Combining the pipelined processor with the cache would also take us a step closer to how computers are designed in industry, thus fulfilling the overall objective of this class. In order to better understand the working of the baseline and alternative implementations, our team evaluated each cache on three different types of loops along with our own subroutines which were designed to tease out the differences between the two implementations. For the three loops provided, our results showed that the performance between the two caches stayed the same for the 1D loop while the direct mapped and two-way set-associative outperformed the other for the 2D and 3D loops, respectively. Contrary to our initial intuition, the alternate design performed worse than the direct mapped cache in the 2D loop because of the *Least Recently Used* eviction policy that we follow with the set-associative cache. While LRU can prove to be beneficial for AMAL with certain program structures, the one used in the 2D loop for the evaluation section for this lab tends to evict the data that is gonna be accessed in the second pass over the array, thus increasing the overall miss rate and consequently the AMAL. However, the performance benefit of the alternate version becomes apparent in the 3D loop where there is almost a 5X improvement in miss rate and 3X improvement in AMAL as compared to the direct mapped cache. Given that the two designs had the exact same statistics on the 1D loop, it could be safely assumed that associative cache's slightly longer critical path did not factor into the overall execution time. Since AMAL = Hit Latency + (Miss Rate * Miss Latency), equal AMAL for the 1D loop suggests that both the designs had the same Hit/Miss Latency since the Miss Rate was the same.

## 2. Baseline Design

For the baseline design we implemented a direct-mapped, write-back, write-allocate cache. The cache is made up of 16, 16-byte cache lines resulting in a total capacity of 256 bytes. The cache can also be used in a bank of caches. In this baseline design, each memory address can only be mapped to one particular cache line, which is determined by the address of the memory request. Furthermore, the write-back, write-allocation means that on a write miss, the cache will first read the data from memory, storing it in the corresponding cache line and then overwriting the new data into the same cache line. This means on a write miss the cache line is immediately dirty. The cache consists of an array for storing the memory tags currently in the cache, as well as an array for storing the actual data corresponding to those tags. The datapath can be seen in **Figure 1**, and the FSM can be seen in **Figure 2**.

The functionality of the direct-mapped cache is quite simple. When a request is sent to the cache, the tag in the cache line corresponding to the address of the request is checked against the tag of the memory request. If they match the request is a hit and no refill/eviction from the cache will occur. On a write, the data is written into the data array to the cache line determined by the request address, and that line is marked 'dirty'. On a read, the data is read from the data array out of the cache line determined by the request address. For cache misses there are two cases to consider. If the cache line is not dirty (it's data is the same as the data stored in the main memory location) a refill is required. On the other hand, if the cache line is dirty, the line will need to be evicted before a refill can occur. For a refill, a read request is sent to the main memory and then the response data is written to the data array and the tag is written to the tag array, hence overwriting the previous entry. Following the refill, the read or write progresses just like previously described. If an evict is required, the data stored in the conflicting cache line is first sent to the main memory in a write request. This updates the value in the main memory that has been changed in the cache. Following the eviction, the refill proceeds as just described and finally the actual read or write to the cache line that was refilled.

Due to the single location for a memory address to map to, this design is a good choice for a benchmark. There are clear advantages to adding more ways since each way added is expected to reduce the number of conflict misses. By keeping our cache size and cache line width the same we expect the same number of compulsory and capacity misses. This means our baseline can give us an accurate measurement of the reduction in conflict misses in alternative designs. This baseline design also utilizes lots of design patterns and principles. First, it utilizes hierarchy extremely well, a key idea in modern memory systems. The cache and memory request/response interfaces allow for unlimited hierarchy. For instance, another cache could be in the layer above or below and the module would not need to know these details. The data would simply be passed and replaced up and down the hierarchy through the interfaces. Like all other labs, the baseline also demonstrates modularity and encapsulation by using input/output parameters to

interface with the module and hiding the actual design of the module from the caller. Finally, the baseline design also uses regularity to allow for the caller of the module to create a bank of these caches at the size of their choosing.

## 3. Alternative Design

For the alternative design we implemented a two-way set-associative, write-back, write-allocate cache with 8 lines per way, each 16-bytes wide. This results in the same capacity as the baseline design: 256 bytes. The biggest difference between the base design and the alternative design is the addition of a new way, which has its own tag and data array. This allows for each memory address to be mapped to two locations in the cache. This provides a big advantage when one of the ways is filled and a new tag is being accessed as it can be placed into the other way, resulting in no eviction of the other tag. This would hopefully lead to a hit the next time the tag in the first way is accessed instead of a miss. The datapath for the alternative design can be seen in **Figure 3**, and the control FSM is the same as the baseline design in **Figure 2**.

The first modification to the base design was to add another tag array and another data array. Furthermore, two more arrays were needed to store the valid and dirty bits of the new way. Next, to determine if there was a cache hit, both tags (one from each way) of the cache corresponding to the request address need to be checked to see if there's a hit. For this we used two separate comparators and then ANDed the results. By having two separate comparators we could keep track of which way had the hit and that allowed us to read/write to the appropriate way. Also, for requests that require a refill or eviction we had to decide which way should be refilled/evicted. For this we are using an LRU replacement policy. We decided to use a single bit, 8 entry register file to keep track of the most recently used way for each cache index since it produced minimal overhead. A '1' being stored in the register file meant way 1 was most recently used and way 0 should be overwritten. A '0' being stored in the register file meant way 0 was most recently used and way 1 should be overwritten. This design choice is as minimal as possible since during a write/read the LRU register file is written parallel to the write of data to a cache line so it should not add much time to the cycle time if any. Also, on the tag check the LRU register file is read in parallel to the read of the tag arrays, once again, likely not adding any time to the cycle time. In addition to keeping track of the most recently used way we had to add muxes to select which data array to read from and which tag array to read from. This was extremely easy to implement thanks to our design choice to use two separate comparators and our use of a LRU register file. On a cache read hit, the data mux select is simply set to the way which had its comparator evaluated to true. On an eviction, the tag mux is simply set to the inverted value from the LRU register file. For example, if the register file bit is 0 (we most recently used way 0) we should select the way 1 tag array as the tag we are evicting, and if the register file bit is 1 (we most recently used way 1) we should select the way 0 tag array as the tag we are evicting. The LRU register file bit is also used during an eviction to set the select of the data mux for a similar reason as just described. As can be seen the design choices on the hit logic and also the LRU tracking didn't only provide the functionality they were required to, but they also provided an easier implementation of the other modifications that were required for the alternative design.

Similar to the baseline design, the alternate design also incorporated many different common design principles in computer architecture. The first and most important design principle incorporated is hierarchy. Just like the baseline design, the two-way associative cache can be used in a hierarchy of different memories as long as they all use the same request/response interface. For example, if a direct cache like the baseline design was above the two-way set associative cache from the alternate design in the hierarchy, on a direct mapped cache read miss the direct mapped cache would send a read request via its memory request interface to the two way set associative cache's cache request interface. If the two way set associative cache had a hit on that request it could return the value back up to the direct mapped cache via it's cache response interface to the direct mapped cache's memory response interface. If the two way set associative cache missed on the request from the direct mapped cache, then it too would have to send a memory request to the memory module below it in the hierarchy. The values in different memories can be passed up and down the hierarchy regardless of the number of memory systems in the hierarchy. In addition to hierarchy, the alternative design also incorporates modularity and encapsulation by using input/output parameters to interface with the module and hiding the actual design of the module from the caller. Finally, the alternative design also uses regularity just like the baseline to allow for the caller of the module to create a bank of these caches at the size of their choosing.

## 4. Testing Strategy

Our overall testing strategy for this lab was to use both directed and random test cases. Furthermore, all of the test cases only needed to be written once because they are shared across the functional-level model, the baseline design, and the alternative design. For each test case, we used the provided req and resp Python helper functions to generate memory request messages and the expected memory response messages. The memory response messages also include a test field, which is set to 1 for a cache hit and 0 for a cache miss. With the knowledge of both cache designs, we were able to write directed test cases that had the same behavior in both designs and some test cases that had different behavior in the different implementations. Therefore, we had three different test case tables: (1) generic tests that had the same behavior in both cache designs and tests specifically designed for (2) the direct-mapped cache and (3)

the two-way, set-associative cache. In addition, because it is difficult to develop white-box ad-hoc tests on the FL model, we also used a design-for-test (DFT) approach, which means some features are added to our design only used for testing purposes. Using this approach, we added an init transaction to our design that writes to the cache based on the tag and index bits of the address. This init transaction must be the very first transaction in a test case and proved useful in the beginning test cases where we check basic functionality of the cache. To summarize, in order to prove the functionality of both cache designs, we used directed tests, random delay tests, and random value tests.

The first kind of testing we implemented was directed test cases. We used directed tests in order to exercise all paths in our datapath as well as all states and state transitions in our FSM. To start, in each test case table, we were provided with a basic test case for the read hit path for clean lines. Next, we created test cases for each of the suggestions in the lab handout including:

- 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 the entire cache
- Conflict misses
- Capacity misses
- LRU replacement policy by filling up a way
- Tests for corner cases in the alternative design

For each of these, we either wrote one test case that has the same expected behavior for both caches and belonged in the generic test case table or two slightly different test cases with behavior dependent on the cache design and in the corresponding cache test case table. For example, for the miss with refill and no eviction case, we only created one generic test. In this test case, and several others, we also use the provided helper function to load data into the test memory prior to running the test. This allowed us to test the cache refill path in our datapath and FSM without having to worry about the evict path. Lastly, for the last two test cases above, we only needed to create tests for the set-associative cache since they deal with specifics of the alternative design. The corner cases that we tested in the alternative design included a cache miss when there is an empty way and evicting from both ways.

Next, we also wrote directed test cases to be sure our cache designs could be used in a four-bank cache organization in the next lab. For the most part, we simply changed the nbank parameter from 0 to 4 in the test case table. This only worked because the test cases had the same expected behavior when the bank bits were or were not ignored. To fully test that our cache worked with 4 banks, we created two test cases: one that resulted in a hit in the correct implementation, but a miss in the incorrect implementation and vice versa.

Another type of testing we used was random delay tests. The reason we included random delay tests was to confirm that we correctly implemented the val/rdy protocol used for memory and testing. To implement these test cases, we reused most of our existing test cases and simply changed the test memory's stall probability, test memory latency, source delay, and sink delay parameters in the test case table. Specifically, we set the stall probability to 0.5 and the other three parameters to a random value between 0 and 5. By reusing test cases that we confirmed were passing with no delays, we knew if any of the random delay tests failed, it was due to an error in our implementation of the val/rdy protocol.

Then, we included random value tests. This allowed us to test a large number of both normal cases and corner cases, saving us a lot of time in typing them all out. We created two test cases, one for the direct-mapped cache and another for the set-associative cache for each of the suggestions in the lab handout noted below:

- Simple address patterns, read request type, with random data
- Simple address patterns, write 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 these random tests was difficult because the correct output, or data from a memory read response, can change if there is a write memory request prior to the read. To keep track of this, we created a reference memory in each of these test cases that was updated during a write request or consulted during a read request. Additionally, we kept track of the cache memory during each test sequence and used it to figure out if the test field in the response message should indicate a cache hit or miss. Finally, we combined the random value tests with random delays to further test our val/rdy protocol.

Through this combination of testing, we are confident that our implementation of both cache designs is correct. The first set of directed tests confirmed we could pass specifical general and corner cases of memory request inputs. Next, the directed tests with nbank set to 4 validated that our design would function correctly in a four-bank cache organization. Furthermore, the random value tests allowed us to test a much larger amount of inputs and further confirm our belief in the cache functionality. Lastly, the random

delay tests on both our directed and random value tests helped us prove that the val/rdy protocol was incorporated correctly. Hence, since we pass all of the tests described above, we believe our designs are fully functional.

## 5. Evaluation

After comprehensively testing out our two implementations, we ran the provided evaluation benchmarks on the two cache designs to critically analyze the performance of each under different program structures. As is evident in **Figures 4** and **5** below, *Loop 1D* performed equally well on both implementations while the base design and alternate design performed better on *Loop 2D* and *Loop 3D*, respectively. *Loop 1D* is a simple loop that does a single pass over an array of 100 elements. Since the cache is only 64 * 4 bytes in total capacity, after the 64th array element, the first cache lines start getting evicted from the cache. This is true for both the designs and hence a similar result follows. Things get more nuanced in *Loop 2D* where there are 5 passes over the same array instead the 1 in *Loop 1D*. **Figures 6** and **7** show the difference in the contents of the two caches at the end of a single pass of the array, just before the same array is going to be accessed again. In the case of the direct mapped cache, it turns out that the green cache lines do not get replaced by the cache at all while the red lines switch in and out. On the first pass of the array, there are 25 (1 Miss, 3 Hit) sequences due to compulsory misses and spatial locality. On each of the next four passes, there are only 18 (1 Miss, 3 Hit) sequences due to conflict misses and then 7 (4 Hit) sequences, which correspond to the green lines, thanks to both temporal and spatial locality. This however, is not the case with the set associative cache where due to the LRU replacement policy, the cache evicts those lines that would have otherwise given a hit. This is shown in **Figure 7a** and **7b**. The red lines get evicted now due to those lines not being the most recently used and thus result in **conflict** misses the next time the corresponding words get accessed. *Loop 3D* is designed to highlight the strength of an associative cache in certain program structures. *Loop 3D* only accesses the 0th word of a cache line and the 16 words are placed in such a way in memory that each half of the array occupies the same 8 lines in the direct mapped cache. This is shown in **Figure 8** where one can see why all memory accesses with the direct mapped cache result in misses. Each half of the array overwrites the other in cache. Thus, we see that all memory accesses are a conflict miss resulting in a very high AMAL. On the other hand, as seen in **Figure 9a** and **9b**, the words accessed in *Loop 3D* fit perfectly in the associative cache. Thus there are **no conflict** misses, only **compulsory** misses resulting from the first access to these words. Once the array is passed over once, there are no more misses in the subsequent passes.

Along with the evaluation patterns provided, we added four more based on programs mentioned in class where spatial and temporal locality are optimized. The first pattern doubles a matrix array by iterating over column then row. The second pattern does the same, but iterates over row then column, which improves the spatial locality. For both of these patterns, the baseline design and alternative design achieved the same miss rate and same AMAL. However, the miss rate dramatically decreased by a factor of 4, from 0.5 in the first pattern to 0.125 in the second pattern. Although both cache designs had the same performance on both of these patterns, which is what we expected, this part of our evaluation did prove the benefits of having optimized code structure. Our next two evaluation patterns showed the effects of improving temporal locality on both cache designs. The first of these patterns had two loops reading and writing the same data, while the second pattern uses just one loop to do the same which improves temporal locality. Similar to the spatial locality patterns, both cache designs had the same miss rate on both patterns, and the miss rate was slightly better on the pattern with better temporal locality. Yet, the AMAL for each design was slightly different for both patterns. For the program with poor temporal locality, the base design had a minimally lower AMAL due to a lower number of cycles. On the other hand, the alternative design performed marginally better on the pattern with improved temporal locality. This is because there are cases when one design will have a dirty cache line needed to be evicted when the other design does not. As a result, the design that needs to evict has to take extra cycles to correctly handle the miss and update memory. Overall, the evaluation patterns we added illustrated how the performance of both cache designs changes based on the amount of spatial or temporal locality.

From the evaluation loops provided to us, we learned that different cache designs perform optimally on different programs depending on the nature and pattern of memory access. There is no one size fit all and higher associativity does not necessarily improve performance. That said, **Figure 4** tends to suggest that when a two-way set-associative cache is compared to a direct-mapped cache of the same capacity, the associative cache would more or less perform as well as the direct mapped cache. On the other hand, certain program structures can really dent the AMAL of the direct-mapped cache. While designing processors and caches, it is important to keep in mind the trade off of increasing associativity. While a two-way set-associative cache did not impact cycle time as much, we expect the critical path and cycle time to increase as we increase the associativity of the cache. In addition, higher associativity also leads to more logic (more *mk_adder* and *tag_match*) to determine in which way the hit has occurred. This in turn leads to higher cost, area used, and energy consumption which have ramifications in an industry setting. As we move onto the final lab where we will combine our pipelined processor and cache memory to create a multicore system, our cache performance will be an important factor and we will certainly use the knowledge we gained during the evaluation process.

**Figure 1:** Baseline Design Datapath

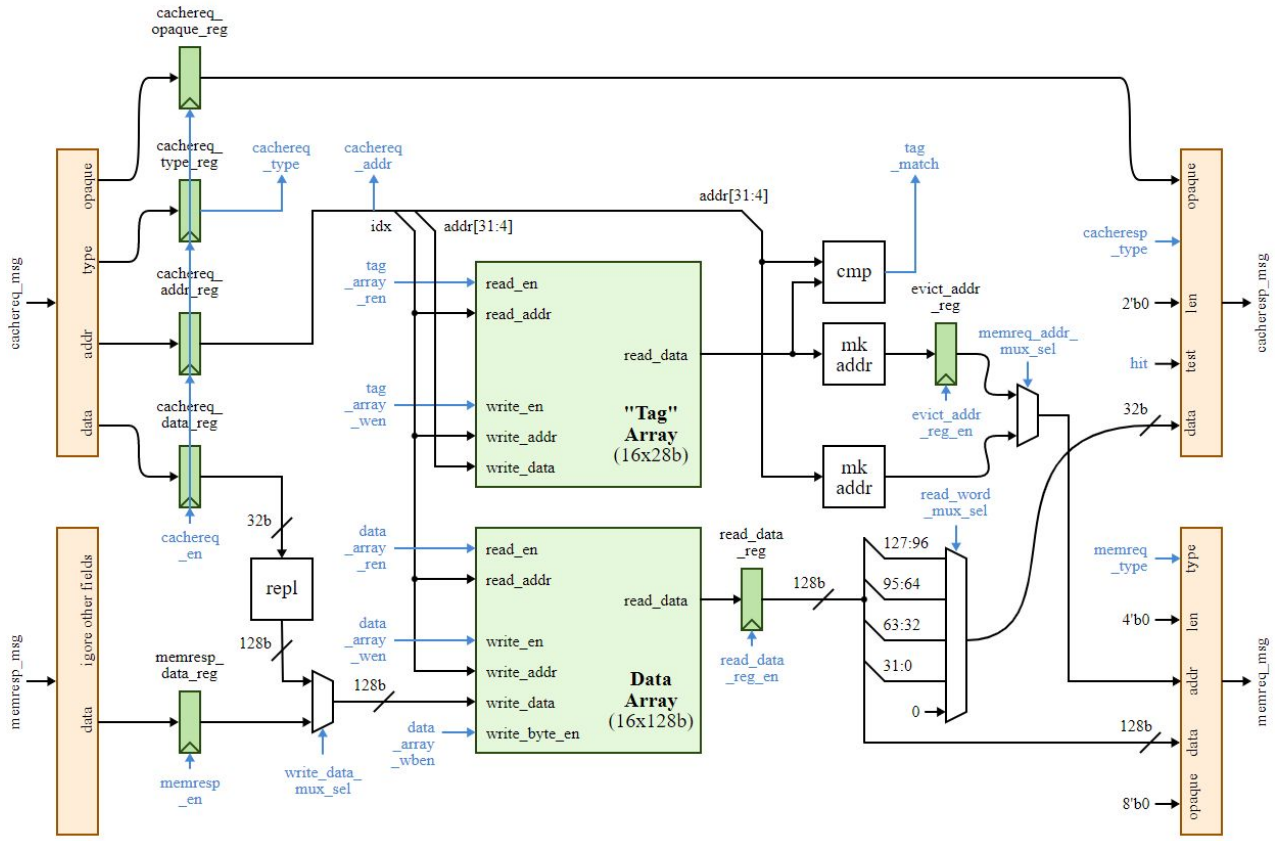


**Figure 2:** Baseline Design Control Unit

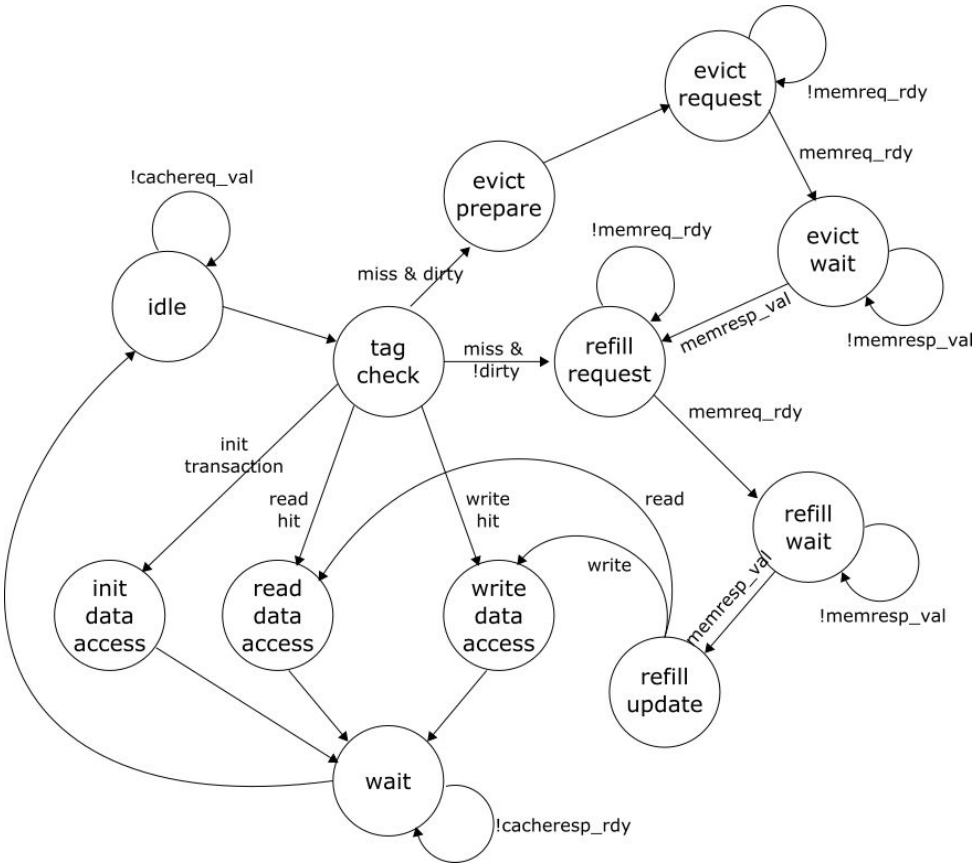

**Figure 3:** Alternative Design Datapath

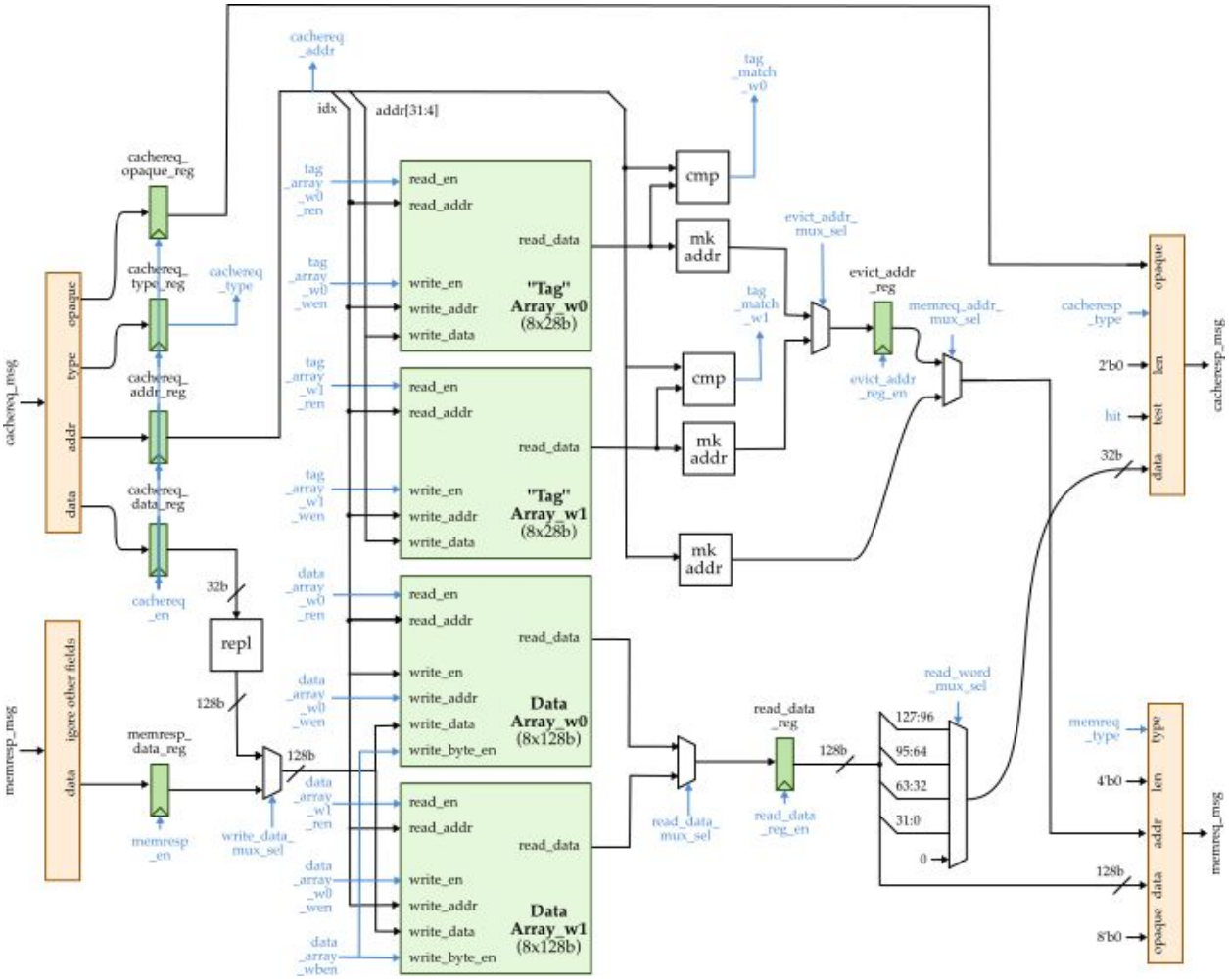


**Figure 4:** AMAL for Loop Patterns

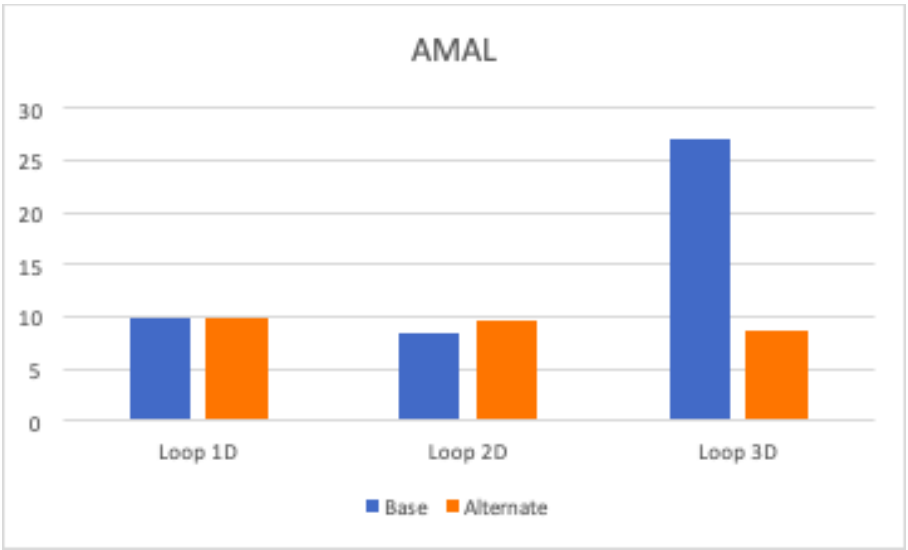

**Figure 5:** Provided Evaluation Results

**Direct Mapped Evaluation Results**

| | # Cycles | # Requests | # Misses | Miss Rate | AMAL |
|---|---|---|---|---|---|
| **Loop 1D** | 976 | 100 | 25 | 0.25 | 9.76 |
| **Loop 2D** | 4232 | 500 | 97 | 0.194 | 8.464 |
| **Loop 3D** | 2161 | 80 | 80 | 1 | 27.01 |

**Two-Way Set-Associative Evaluation Results**

| | # Cycles | # Requests | # Misses | Miss Rate | AMAL |
|---|---|---|---|---|---|
| **Loop 1D** | 976 | 100 | 25 | 0.25 | 9.76 |
| **Loop 2D** | 4232 | 500 | 125 | 0.25 | 9.75 |
| **Loop 3D** | 2161 | 80 | 16 | 0.2 | 8.61 |

**Figure 6:** Contents of the direct mapped cache after the first pass of the array

| | 11 | 10 | 01 | 00 |
|---|---|---|---|---|
| **0000** | ... | ... | ... | **0xf0** |
| **0001** | ... | ... | ... | **0x100** |
| **0002** | ... | ... | ... | **0x110** |
| **0010** | ... | ... | ... | **0x120** |
| **0011** | ... | ... | ... | **0x130** |
| **0100** | ... | ... | ... | **0x140** |
| **0101** | ... | ... | ... | **0x150** |
| **0110** | ... | ... | ... | **0x160** |
| **0111** | ... | ... | ... | **0x170** |
| **1000** | ... | ... | ... | **0x180** |
| **1001** | ... | ... | ... | **0x90** |
| **1010** | ... | ... | ... | **0xa0** |
| **1011** | ... | ... | ... | **0xb0** |
| **1100** | ... | ... | ... | **0xc0** |
| **1101** | ... | ... | ... | **0xd0** |
| **1111** | ... | ... | ... | **0xe0** |

**Figure 7a:** Contents of 2-Way Set Associative (Way 1) after the first pass of array

| LRU (X represents line was least recently used) | | 11 | 10 | 01 | 00 |
|---|---|---|---|---|---|
| | 000 | ... | ... | ... | **0x100** |
| X | 001 | ... | ... | ... | **0x110** |
| X | 010 | ... | ... | ... | **0x120** |
| X | 011 | ... | ... | ... | **0x130** |
| X | 100 | ... | ... | ... | **0x140** |
| X | 101 | ... | ... | ... | **0x150** |
| X | 110 | ... | ... | ... | **0x160** |
| X | 111 | ... | ... | ... | **0x170** |

**Figure 7b:** Contents of 2-Way Set Associative (Way 2) after the first pass of array

| LRU | | 11 | 10 | 01 | 00 |
|---|---|---|---|---|---|
| X | 000 | ... | ... | ... | **0x180** |
| | 001 | ... | ... | ... | **0x90** |
| | 010 | ... | ... | ... | **0xa0** |
| | 011 | ... | ... | ... | **0xb0** |
| | 100 | ... | ... | ... | **0xc0** |
| | 101 | ... | ... | ... | **0xd0** |
| | 110 | ... | ... | ... | **0xe0** |
| | 111 | ... | ... | ... | **0xf0** |

**Figure 8:** Contents of the direct mapped cache at the end of the first pass of the array

| | **11** | **10** | **01** | **00** |
|---|---|---|---|---|
| **0000** | **...** | **...** | **...** | **0x100** |
| **0001** | **...** | **...** | **...** | **0x110** |
| **0010** | **...** | **...** | **...** | **0x120** |
| **0011** | **...** | **...** | **...** | **0x130** |
| **0100** | **...** | **...** | **...** | **0x140** |
| **0101** | **...** | **...** | **...** | **0x150** |
| **0110** | **...** | **...** | **...** | **0x160** |
| **0111** | **...** | **...** | **...** | **0x170** |
| **1000** | | | | |
| **1001** | | | | |
| **1010** | | | | |
| **1011** | | | | |
| **1100** | | | | |
| **1101** | | | | |
| **1110** | | | | |
| **1111** | | | | |

**Figure 9a:** Contents of 2-Way Set Associative (Way 1) at the end of the first pass of the array

| LRU (X represents line was least recently used) | | 11 | 10 | 01 | 00 |
|---|---|---|---|---|---|
| | 000 | ... | ... | ... | **0x00** |
| | 001 | ... | ... | ... | **0x10** |
| | 010 | ... | ... | ... | **0x20** |
| | 011 | ... | ... | ... | **0x30** |
| | 100 | ... | ... | ... | **0x40** |
| | 101 | ... | ... | ... | **0x50** |
| | 110 | ... | ... | ... | **0x60** |
| | 111 | ... | ... | ... | **0x70** |

**Figure 9b:** Contents of 2-Way Set Associative (Way 2) at the end of the first pass of the array

| LRU | | 11 | 10 | 01 | 00 |
|---|---|---|---|---|---|
| X | 000 | ... | ... | ... | **0x100** |
| X | 001 | ... | ... | ... | **0x110** |
| X | 010 | ... | ... | ... | **0x120** |
| X | 011 | ... | ... | ... | **0x130** |
| X | 100 | ... | ... | ... | **0x140** |
| X | 101 | ... | ... | ... | **0x150** |
| X | 110 | ... | ... | ... | **0x160** |
| X | 111 | ... | ... | ... | **0x170** |

**Role and Task Table**

| **Carly Swetz** | **Raghav Kumar** | **Kyle Betts** |
|---|---|---|
| RTL Verification Engineer | RTL Design Engineer | RTL Design Engineer (architect) |
| ● Wrote all direct tests, random delay tests, and random value tests for the generic test case table, the direct-mapped test case table, and the set-associative test case table<br>● Brainstormed and created test cases to ensure our cache design would work in a four-bank cache organization<br>● Made sure that all tests were passing on the FL model, the baseline design, and the alternative design<br>● Helped debug the baseline and alternative designs when certain test cases were failing<br>● Wrote the Testing section of the lab report where the overall testing strategy is summarized<br>● Added the additional evaluation patterns with different amounts of spatial and temporal locality and a mix of reads and writes<br>● Contributed to Evaluation section of lab report<br>● Prepared lab report for submission | ● Implemented the initial base datapath<br>● Implemented the initial base control structure<br>● Implemented few of the basic hit paths on the base version<br>● Evaluated the three loops provided<br>● Did a detailed analysis of the provided evaluation benchmarks, mapping out the cache contents over time<br>● Made tables and graphs elucidating the evolution of the cache contents in the eval section<br>● Wrote part of the eval section on report pertaining to given benchmarks<br>● Added the additional evaluation patterns with different amounts of spatial and temporal locality and a mix of reads and writes along with Verification Engineer<br>● Wrote Introduction section in the lab report | ● Implemented the base datapath and the base control FSM<br>● Degugged errors in the Base design with help for the verification engineer<br>● Helped discover issues in our base design testing by walking through the line tracing<br>● Designed the datapath and control FSM for the alternate design<br>● Implemented the alternate datapath and the alternate control FSM<br>● Degugged errors in the alternate design with help for the verification engineer<br>● Helped discover issues in our alternate design testing by walking through the line tracing<br>● Created the line tracing for both the alternate and base designs to aid in debugging<br>● Created the alternate design path SVG<br>● Wrote up the sections baseline design and alternate design for the lab report<br>● Provided insight to the evaluation results based on knowledge on our designs |