# ECE 4750 - Lab 3 Report

Sabrina Herman (sh997), Yoon Jae Oh (yo82), Benjamin Xing (byx2)

## Section 1: Introduction

The purpose of this lab was to gain experience with topics taught in class such as basic memory system design, complex finite-state-machine cache controllers, agile design methodologies, and design patterns through designing two types of FSM caches. The lab incorporates many important themes in computer architecture like analyzing cache performance through hit and miss rates, and therefore relates to lecture materials as well as industry practices. We designed a control unit and datapath with a split design pattern, building off of what we learned in labs 1 and 2. We used an incremental development design and testing methodology by beginning with a baseline design before moving on to an alternate design. These computer architecture themes are relevant not only to this class, but to how computer architecture, design, and verification are implemented in industry.

The lab taught us different types of cache designs for dealing with data storage and retrieval through the implementation of two different cache designs. In the baseline design, we designed a direct-mapped cache, where every cache line can only be placed in a single location. In the alternative design, a two-way set associative cache was designed, in which every cache line can be placed in one of two locations in the cache. Both caches used a write-back, write-allocate for handling write misses, meaning that writes are only written to memory when that cache line needs to be evicted from the cache, rather than writing through to memory every time we write to the cache. For both designs, we created components in the datapath, set values in the control unit, and edited the parent module which together incorporated the cache designs. We learned that in a FSM cache design, it is important to carefully determine state transitions and state outputs in order to pass and output accurate data. It is also very important to carefully manage what data is kept in the cache. Understanding how the parent module interfaced between the datapath, the control unit, the memory, and the test source/sink was essential for completion of the lab, and taught us design patterns including message interfaces between the processor and memory, the control/datapath split, and cache design.

The alternative design is more compelling than the baseline design for larger caches because it can reduce the miss rate by reducing the number of conflict misses. This is due to the fact that the baseline (direct-mapped) design only allows cache lines to be placed in a single location whereas the alternative (two-way set associative) design allows cache lines to be placed in one of two locations. This reduces the number of conflict misses because if there is already a cache line in one of the ways of the set-associative cache, the incoming cache line can just be put in the other way, avoiding a conflict miss. Although the set-associative cache may require more area and energy due to increases in required hardware and energy, it can be well worth the costs due to the hit rate improvements that it provides. The set-associative cache's lower miss rate means that it doesn't have to access the main memory as much as the baseline design, which improves the set-associative cache performance when the hit rate is critical. Therefore, even though the alternative design requires more area and energy, its higher hit rate outweighs those increases. In general, increasing the number of ways in a set-associative cache reduces the number of conflict misses and in turn optimizes for avoiding miss penalties. Direct-mapped caches, on the other hand, optimize hit latency and area/power requirements for smaller caches, such as those directly below the processor level.

## Section 2: Baseline Design

The baseline design is a direct-mapped, write-back, write-allocate cache with a total capacity of 256 bytes, 16 cache lines, and 16 bytes per cache. The baseline design utilizes the control/datapath split design and the control unit uses an FSM. This decomposition is an example of the modular design principle, which is an approach that subdivides a system into smaller modules that can be independently created. This method is utilized because it allows the different parts to be independently created and used in different systems, which is effective as well as efficient as multiple copies of the same sub-system do not need to be created in different systems.

Our control unit FSM is similar to our FSM for the first lab, with always blocks for next state logic and state transitions, but we decided to use a control signal table, similar to the cs table in the second lab, for our output logic. The control signal table was convenient since there were many output signals required for many states, and this information was displayed cleanly through the table, and was therefore easier for debugging incorrectly set signals. To handle valid and dirty bits associated with the cache lines of our datapath, we implemented register files in our control unit. One register file was for valid bits, the other register file for dirty bits.

Our datapath for our cache was relatively simple. The main components of datapath were the tag and data arrays. Other components were registers and muxes which were needed for correctly storing and moving information into and out of the arrays. The baseline datapath, as shown in Figure 1, was implemented all at once, since it was relatively simple and most paths needed to be in place for basic transactions. However, we used an incremental design approach for our control unit, because it was far more complicated and easier to break down into more basic transactions, since certain operations only require some of the states of our final

FSM design. The first basic transactions that were tested were the initial transactions, the read hit transaction, and the write hit transaction. These operations were more basic because they do not require access to memory and only use a few of the FSM state to execute. Our datapath exhibits the principles of modularity and encapsulation as the tag and data SRAMs, along with the valid and dirty register files, did not affect the rest of the datapath - they could simply be inserted as self-governing modules.

The control unit, as shown in Figure 1, consists of 12 FSM states: idle (I), tag check (TC), init data access (IN), read data access (RD), write data access (WD), wait (W), evict prepare (EP), evict request (ER), evict wait (EW), refill request (RR), refill wait (RW), and refill update (RU). The idle state receives the incoming cache request, places it in the input registers, and moves to the TC state once the cache request is valid, otherwise it stays in the idle state. The TC state checks the tag and moves to the RD state on a read hit, the WD state on a write hit, IN state on an initial transaction, the EP state on a miss and a dirty cache line, and the RR state on a miss and a non-dirty cache line. The IN state immediately writes to the appropriate cache line and then moves to the W state. The RD state reads from the appropriate cache line and then moves to the W state. The WD state writes to the appropriate cache line and then moves to the W state. The W state waits for the cache response to be ready and once it is, moves to the I state. The EP state reads the tag and data, prepares the eviction message and then moves to the ER state. The ER state makes a request to memory to write the evicted cache line and moves to the EW state once the memory request is ready. The EW state waits for the memory response and moves to the RR state once the memory response is valid. The RR state makes a request to memory to write the evicted cache line and moves to the RW state once the memory request is ready. The RW state waits for memory response and moves to the RU state once the memory response is valid. The RU state writes the response to the victim cache line and moves to the RD state if the type is a read or to the WD state if the type is a write.

This design is a good baseline for comparison because a direct-mapped cache is a basic cache without optimizations that try to reduce the miss rate. These optimizations require greater complexity in our control logic, as well as more hardware in our datapath. Therefore, this design provides a good baseline to build on and optimize later in our alternative design in order to improve the hit rate. Being able to compare other designs to a direct-mapped cache allows us to easily evaluate other designs and draw conclusions about the hit and miss rates.

## Section 3: Alternative Design

The alternative design is a two-way set associative, write-back, write-allocate cache with the same capacity and cache line size as the baseline design. Compared to the direct-mapped cache, the set-associative cache results in a larger hit rate since for any given index there are two different ways where the tag can be stored. At the same time, however, associativity comes at the cost of additional hardware requirements, more complex control logic, and a potentially greater latency. Specifically, a set-associative design requires separate SRAM modules for the Way 0 and Way 1 tag arrays, and it requires control and status signals to determine which of the two ways produces a hit. Additionally, in the case of a miss, there needs to be a replacement policy to decide which of the two ways to overwrite. In this case, we implement a least-recently-used replacement (LRU) policy to choose between the two ways during eviction as this policy is empirically demonstrated to have good performance. Like the baseline design, the alternative design is based on the datapath/control split principle, where the control unit sends signals to the datapath according to the state of the FSM logic.

A key difference between the alternative and the baseline design is that a hit can occur from a tag match in either Way 0 or Way 1, and we need to know which way it occurred in. Therefore, we added a second equality comparator module and an additional status signal for the extra tag check that must be performed. We also added two more register files in the control unit for Way 1's valid and dirty bits. The way-select signal, which is generally used as the part of the data array read or write address, handles both hits and misses (but not evictions). Since it depends on the values of the tag matches, valid, and dirty bits, we compute the logic for way-select in the control unit and output it to the datapath module as a control signal. With a two-way set-associative cache, there are now only three index bits in an address to select one of eight entries in each tag array. Since the data array remained the same size at 16 entries while the tag arrays now each store only 8 entries, we decided it would be organizationally beneficial to simply assign the first eight entries in the data array to Way 0, and the last eight entries to Way 1. This structure was simple to implement since the 3-bit index value could simply be appended to a 1-bit way-select signal that chooses which half of the data array to index into. While organizing the data array as we did was perhaps not strictly necessary, we believe this demonstrates the regularity principle since it provides a direct mapping from the tag arrays to the data array and makes the implementation easier to understand.

To track each cache line's least-recently used way, we use a one-bit register file with 8 entries in the control unit. The entries simply toggle high or low to indicate that Way 1 or Way 0, respectively, holds the least-recently used tag for the cache line at the corresponding index. The LRU information is used in the refill request (RR) state to determine which of the two ways to overwrite. We assign the output of this register file to the victim signal, which feeds into a mux in the datapath. The other input to the mux is the more general way-select signal, which is set depending on what type of transaction is occurring and whether it results in a hit or miss.

The mux select signal depends on whether the transaction will require an eviction; if so, then the victim signal is passed through the mux; otherwise, the way-select signal is passed through. The output of this mux is concatenated with the 3-bit index to index into either the upper half of the data array or the lower half, depending on which way we are working with. The final addition to the datapath is a mux which takes as inputs the outputs of both tag arrays, and selects which one to pass through based on the victim signal. The output of this mux can now be used as the eviction address. We implemented a register file to manage the LRU as an application of modularity and encapsulation, since the register file can be seen as a black box that regulates itself and can easily be inserted into an existing microarchitecture. Since the behavior of the register file had already been provided, it was a logical design choice to include a well-defined hardware structure.

Unlike the alternative designs in the previous two labs, the set-associative microarchitecture in this lab is not necessarily generally better than the baseline direct-mapped microarchitecture. It is, however, optimized for use as a larger cache level, farther from the processor than the direct-mapped cache would be. This is because the set-associative cache generally has a better hit rate than direct-mapped caches (a given index can map to more than one cache line), but at a higher latency and hardware area cost.

**Section 4: Testing Strategy**

We used incremental and test driven methodologies for completing both the baseline and the alternative design. That is, we verified our implementation of each instruction (read hit, write hit, read miss, and write miss) before continuing to the next one by running the provided basic test and a few simple directed value tests we write during the development process. After fully implementing the datapath and control modules, we comprehensively tested each instruction by adding directed test cases for different cache paths. We also added random value testing as well as random delay testing to ensure that the processor could handle all cases and the uncertainty of real-world computation. We combined these directed and random tests into unit test suites for each instruction that not only verified the correctness of the processor in isolated tests but also in sequences of instructions. Throughout the testing process, we use the line trace outputs of the unit tests to locate errors and debug faulty connections and control logic in our modules.

For both designs, it was important to test transactions that would hit/miss based on the cache microarchitecture. We wrote directed tests that specifically target different functions of the datapath and FSM. For example, we tested filling an entire cache line, conflict misses for the same line of the cache, reading/writing to all four words in the cache line, writing to every line of the cache, and capacity misses. To ensure the correctness of these cases, we tailor the index bits of the addresses to be the same for testing evictions and hits, and to be different for filling up the cache. For the alternative design, we write tests to check that we are handling the way selection properly. For example, we had to test writing to the same set in both ways, conflict misses in both ways, filling the entire cache (which uses different addresses than for direct-mapped design), and evicting/refilling the correct way of the cache. We particularly wanted to test writing to the ways in patterns other than a purely alternating pattern by writing to the same address multiple times in a row. Directed tests were very useful in debugging our code as well as ensuring that we could handle edge cases correctly. To make sure that we handled the ready and valid interfaces correctly, we added random stalls and delays to the source, sink, and memory. These tests ensure that our cache will perform correctly in the uncertain environment of real computation.

We also wrote fully randomized tests that simulate cache operation over a sequence of many transactions. We generate random address and data values, and feed these into the Python assembly generation functions. However, to make sure that our microarchitectural implementation performs correctly, we also simulate the cache at a functional level in Python by creating arrays to store the tag and data values, as well as a large array to act as "main memory." We update the values in the Python tag and data arrays for each transaction that occurs and predict a hit or miss based on whether the newly generated tag matches the one that already exists at the corresponding index in the Python tag array. When we must evict to memory or read from memory, we simply index into the memory array using the full 32-bit address. For the alternative design, we implement the least-recently used logic in Python and create separate Python arrays for the two tag arrays to keep track of the contents of the set-associative cache. To ensure that the tests do not pass simply because the possible address space is very large (i.e. the caches will just miss every time and the replacement logic will not be tested), we restrict the address space to only approximately 250 possible locations. We run through 200 iterations of randomized cache transactions to test a balanced mix of hits, misses, and evictions. Along with random delay testing, our random value tests ensure not only that the microarchitecture works in general but also that we did not forget to include any special test cases in our directed tests.

**Section 5: Evaluation**

We chose to evaluate cases that would range in level of optimization, for example highly optimized, somewhat optimized, barely optimized, and not optimized. For example, although we conclude that hit rate is generally higher for a set associative, we do evaluate a special case in which hit rate is higher for direct mapped. The evaluation cases (Table 3) provided include loop-1d which reads an array once. The evaluation indicated that this case performed the same on both cache microarchitectures. This is due to the fact that neither cache can avoid a compulsory miss (the first access to a memory block that must be brought into the cache). The loop-3d case reads the same cache line and for this case, the base performs worse than the alternative. This is because with the direct-mapped, the conflict misses are unavoidable when reading the same cache line whereas the set-associative has different ways that it can choose from, which helps it avoid conflict misses. The loop-2d case reads the same array as the loop-1d case but does so 5 times. For this test, the direct-mapped performed better because the direct-mapped only has one location of choice to look for the particular line whereas the set-associative has to check both ways for the line it wants, making the direct-mapped cache perform better.

In our test random-unique. We write random, yet unique addresses to the cache. The way for the cache to hit is to write to a word already in the cache line of a previous address brought in. This test is very similar to the first test of loop-1d, except that it has no spatial locality since the addresses are random. Even without the guarantee of neighboring values, the cache is still able to hit on words from cache lines already in the cache. This demonstrates that when the cache misses due to compulsory misses, even when there is no spatial locality, the two caches behave the same.

In our test copy-add1-array, we iterate through two arrays. We read the value of the first array, then write that value incremented by one to the next array. This is a good measure for evaluation because it shows how these cache microarchitectures behave in a common function of copying an array. In a direct mapped, the cache will miss every time because the addresses we've chosen have the same index bits, yet different tags since they are writing to different addresses in memory. In the set-associative cache, the fact that these two arrays coincide with the same index bits is not a problem, since there are two locations for the two arrays.

In our test vvadd, we iterate through two arrays, then add the two vectors together into a third array like we did in class with the vector-vector add function. This is a good measure for evaluation because it shows how these cache microarchitectures behave in a common function of adding two arrays together into a third array. In a direct mapped, the cache will miss every time because the addresses we've chosen have the same index bits, yet different tags since they are writing to different addresses in memory, like how it did in the copy-add1-array evaluation,. In the set-associative cache, the fact that these two arrays coincide with the same index bits is now a problem, since there are only two locations for the three arrays. Because this set-associative cache uses a least recently used replacement, there will always be conflicts since it adds array 1 to the cache, then array 2, then array 3. Then it needs the array data in the same order for the next iteration of the loop. As a result, the set-associative has the same miss rate as the direct mapped cache. What is interesting is our cycle time and average memory access latency, which go up in the set-associative cache because the two ways require more cycles to handle evicting the written data.

The area of the alternative design is greater due to increased logic, which is necessary for that design in order to optimize the cache hit rate and decrease main memory access. However, the level of optimization that the alternative design provides is worth the area consumption tradeoff, especially for the applications where a set-associative cache is used most often (e.g. L2 or L3 caches). Therefore, although the alternative design may consume more area due to additional logic, the significant increase in hit rate that it produces makes the extra area consumption acceptable, especially when the alternative is a very long memory latency.

Reducing energy consumption use is a crucial part of reducing operating costs in computer architecture. Comparing our baseline and alternative designs, the alternative design consumes more power by virtue of its slightly more complex datapath and control logic. However, for cases where the set-associative cache is optimized, the number of misses decreases and so does the number of memory accesses as a consequence. As a result, less energy may be consumed overall due to the smaller number of misses and evictions that cause the number of cycles to increase. On the other hand, for applications like the small L1 cache directly below the processor, a direct-mapped cache is sufficient (and the better choice) since its simplicity allows a very low latency for a hit and it can fall back on the lower level caches for a miss.

Overall, the results of the evaluation demonstrate the differences between a direct-mapped and a two-way set associative cache. A direct-mapped cache, much like our baseline design, only allows cache lines to be put in one location. Therefore, the design is not optimized for hit rate because the baseline design misses the opportunity to take full advantage of the cache's capacity. However, as previously discussed, the alternative design optimizes both the hit rate and energy-consumption through increased performance and efficiency, though possibly at the cost of higher latency. Although the area consumption was not decreased with the alternative design compared to the baseline design, the upgrades that the alternative design brings are far more important for supporting higher cache levels than the amount of area that it consumes. Therefore, we can conclude that the alternative design is better optimized for situations where hit rate is critical like the larger caches closer to main memory, while the baseline design is better for small caches closer to the processor.

## Section 6: Additional Diagrams

Figure 1. Baseline Datapath

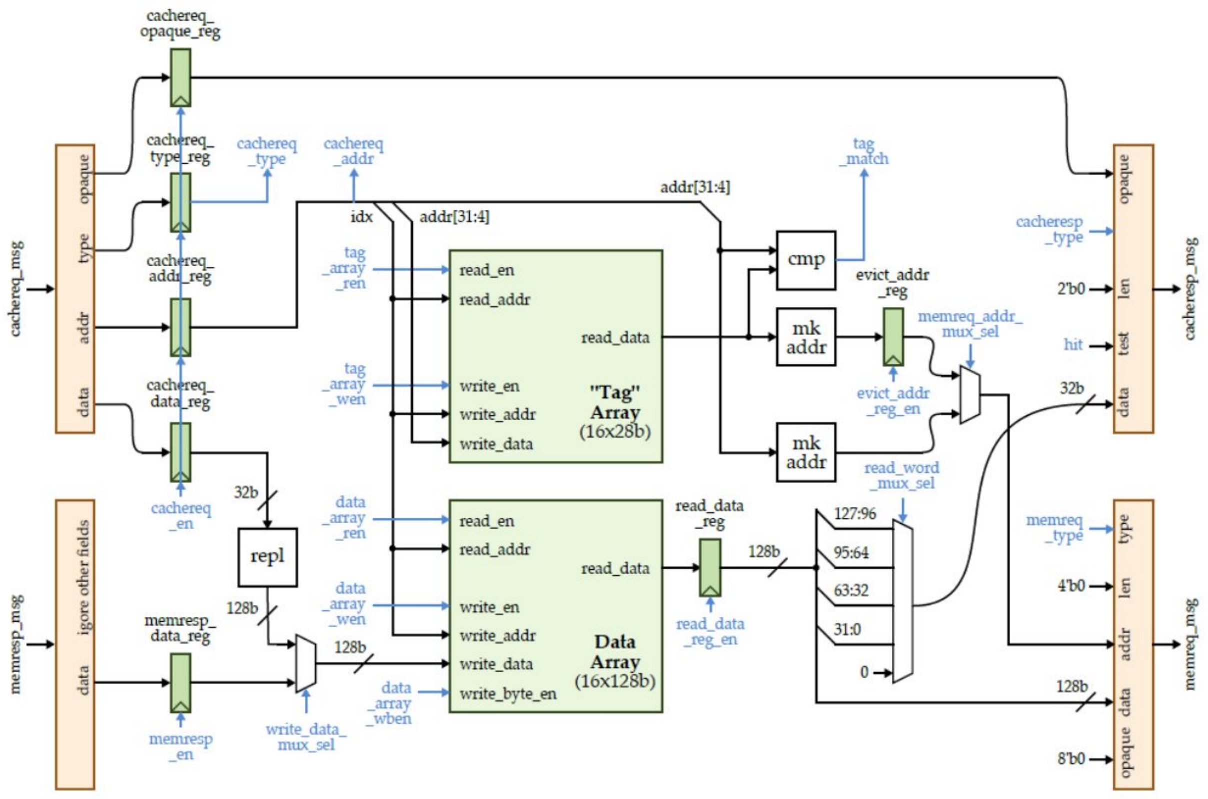

Figure 2. Baseline FSM Control Unit

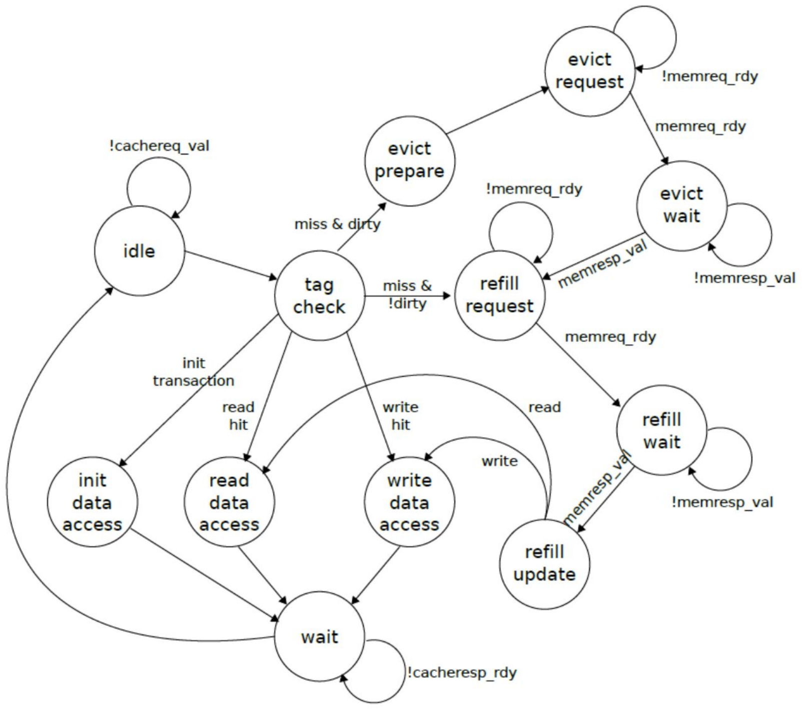

Figure 3. Alternative Datapath

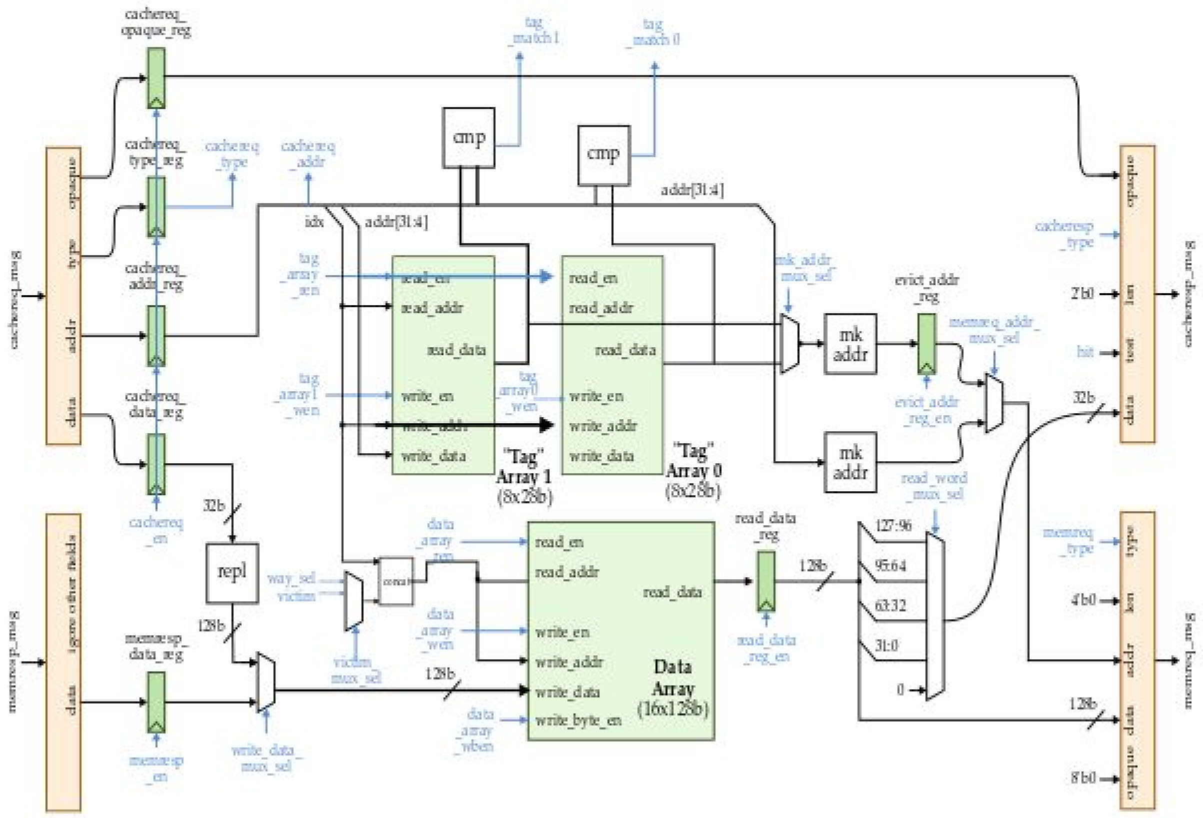

Table 1. Task Assignments

| Lab | sh997 | byx2 | yo82 |
|---|---|---|---|
| Lab 1 | RTL | Verification | RTL (architect) |
| Lab 2 | Verification | RTL(architect) | RTL |
| Lab 3 | RTL(architect) | RTL | Verification |
| Lab 4 | | | |
| Lab 5 | | | |

Table 2. Tasks Done by Each Member

| | sh997 | byx2 | yo82 |
|---|---|---|---|
| Tasks | **Baseline Design:**<br>-Designed datapath<br>-Started control next state and output logic<br>-Debugged Baseline control unit issues<br>**Alternative Design:**<br>-Debugged “way” choosing<br>**Testing:**<br>-Wrote majority of the tests, and came up with specific tests that stressed direct | **Baseline Design:**<br>-Implemented datapatch<br>-Helped with baseline control logic<br>**Alternative Design:**<br>-Designed datapath components<br>-Debugged state transition logic and LRU logic<br>**Testing:**<br>-Wrote fully randomized tests for baseline and | **Baseline Design:**<br>-Put together control table structure<br>-Helped with baseline control logic<br>**Alternative Design:**<br>-Put together datapath<br>-Incorporated way and LRU logic for control unit<br>**Testing:**<br>-Helped look over correctness of tests |

| | | | |
|---|---|---|---|
| | mapped vs set-associative structure<br>**Lab Report:**<br>-Baseline<br>-Testing section<br>-Evaluation | alternative design<br>**Lab Report:**<br>-Alternative section<br>-Testing section | **Lab Report:**<br>-Introduction, Baseline, and Evaluation sections |

Table 3. Evaluation Table

| **Test** | | **Base** | **Alt** |
|---|---|---|---|
| **loop-1d** | Cycles | 976 | 976 |
| | Requests | 100 | 100 |
| | Misses | 25 | 25 |
| | Miss Rate | 0.25 | 0.25 |
| | AMAL | 9.76 | 9.76 |
| **loop-2d** | Cycles | 4232 | 4876 |
| | Requests | 500 | 500 |
| | Misses | 97 | 125 |
| | Miss Rate | 0.194 | 0.25 |
| | AMAL | 8.464 | 9.752 |
| **loop-3d** | Cycles | 2161 | 689 |
| | Requests | 80 | 80 |
| | Misses | 80 | 16 |
| | Miss Rate | 1.0 | 0.2 |
| | AMAL | 27.0125 | 8.6125 |
| **Random-unique address** | Cycles | 1666 | 1689 |
| | Requests | 100 | 100 |
| | Misses | 55 | 56 |
| | Miss Rate | 0.55 | 0.56 |
| | AMAL | 16.66 | 16.89 |

| | | | |
|---|---|---|---|
| **copy_add1_array** | Cycles | 4561 | 1433 |
| | Requests | 128 | 128 |
| | Misses | 128 | 32 |
| | Miss Rate | 1.0 | 0.25 |
| | AMAL | 35.6328 | 11.1953 |
| **vvadd** | Cycles | 6289 | 6473 |
| | Requests | 192 | 192 |
| | Misses | 192 | 192 |
| | Miss Rate | 1.0 | 1.0 |
| | AMAL | 32.7552 | 33.7135 |