Anne Potter (ap674) and Ashley Heckman (agh93)
Lab Report 3
ECE 4750
11/10/22

Lab Report 3

1. Introduction

In this lab, we implemented two different blocking caches. The first implementation, or the baseline design, is a direct-mapped cache. The second implementation, or the alternative design, is a two-way set-associative cache that uses the least recently used (LRU) replacement policy. Both caches are write-back and write-allocate, with the ability to act as a bank in a multi-banked cache. Both caches have 16 cachelines with 16 bytes per cache line, meaning they both have a total capacity of 256 bytes. In the direct mapped cache, each cache line can only be placed in a single location in the cache. In the two-way set-associative cache, each cache line can be placed in one of two locations in the cache, which will generally reduce the number of conflict misses. This lab is intended to give us a deeper understanding of caches, as well as to familiarize us with and give us hands-on experience with the material we have covered in lecture. These topics include the Verilog hardware description language, cache lines, cache associativity, the LRU replacement policy, cache hits/misses, the write-back with write allocate write policy, FSM caches, and the latency insensitive stream interfaces that use the val/rdy micro-protocol. The cache interfaces with the processor from lab two and is the third step towards completing a full multicore system.

2. Alternative Design

Both cache implementations are write-back with write-allocate, with 16 total cache lines. Each cache line holds 16 bytes, or four words. When the cache reads data from main memory, it will read in an entire cache line. For example, if there is a cache request involving address 0x1004, the cache will bring in addresses 0x1000, 0x1004, 0x1008, and 0x100c. Since the caches are write-allocate, if the data we want to access is not in the cache, we must bring that data into the cache. Additionally, since the caches are write-back, if the data is changed, it isn't immediately written back to main memory. Instead, it is kept in the cache with a high dirty bit to indicate that the data in the cache doesn't match the data in main memory. Then, when that data is evicted from the cache, main memory is written with the new value that was already in the cache.

The cache request message is broken up into four fields: data, address, type, and opaque. The data field holds the data that we want to write to main memory or initialize in the cache. The address field holds the address that we want to read, write, or initialize. The type field indicates whether the request is an init, read, or write transaction. Read transactions read the data from memory at the specified address, write transactions write the data to memory at the specified address, and init transactions initialize the cache with the given address and data without reading or writing from main memory. The opaque field is most commonly used to handle out-of-order memory responses. Our cache doesn't support this, so we simply preserve the opaque field and send it to the cache response message; it isn't used to decide anything within the cache.

The address field is further broken down into four fields: index, byte offset, word offset, and tag. The byte offset identifies which byte in the word to read. The word offset identifies which word in the cache line to read or write. It is passed through a decoder unit whose output indicates which bits of the data array should be written to or read; it always outputs the bits that correspond to the given word. The index field is used to determine which set to read or write the data to/from. For example, if the index is zero, the first cache line will be accessed, if the index is one, the second cache line will be accessed, and so on. This field determines which line of the tag and data arrays get read or written. The tag indicates whether the data is already in the cache. If the tag in the tag array at the index from the cache request matches the tag from the cache request, the data is already in the cache, and if the data is valid, it's a cache hit. Otherwise, if the tags are different or the data is invalid, it's a cache miss.

If the data is not yet in the cache or if data needs to be evicted from the cache, the cache needs to send a request to main memory. In the case of the data not yet being in the cache, the cache will concatenate the index from the cache request, the tag from the cache request, and 4'b0000 to make the address for the request. However, if the data in the cache needs to be evicted, the cache will form the address by using the tag that was stored in the tag array at that index.

One thing to note is that the cache request and response send and receive 32 bits of data, but the memory requests and responses and the data array hold 128 bits of data. This is because the cache requests and responses only read or write a single word, but the main memory requests and responses read and write the entire cache line. The cache selects which word to send to the processor by using the word offset to pick which word to read from the data array. On a read request, this word is sent back to the processor by the cache. On a write or init response, the cache simply sends zero back to the processor.

The alternative design has a lot in common with the baseline design. The biggest difference between the two implementations is that the alternative design allows data at a given index to be placed in one of two locations, while the baseline design only allows data to be placed at one location at a given index. Thus, the baseline design has 16 indexes with one possible data location, while the alternative design has 8 indexes with two possible data locations. As a result, the baseline design requires four index bits, while the alternative design requires three index bits. Their address maps are shown in *Figure 1* below.

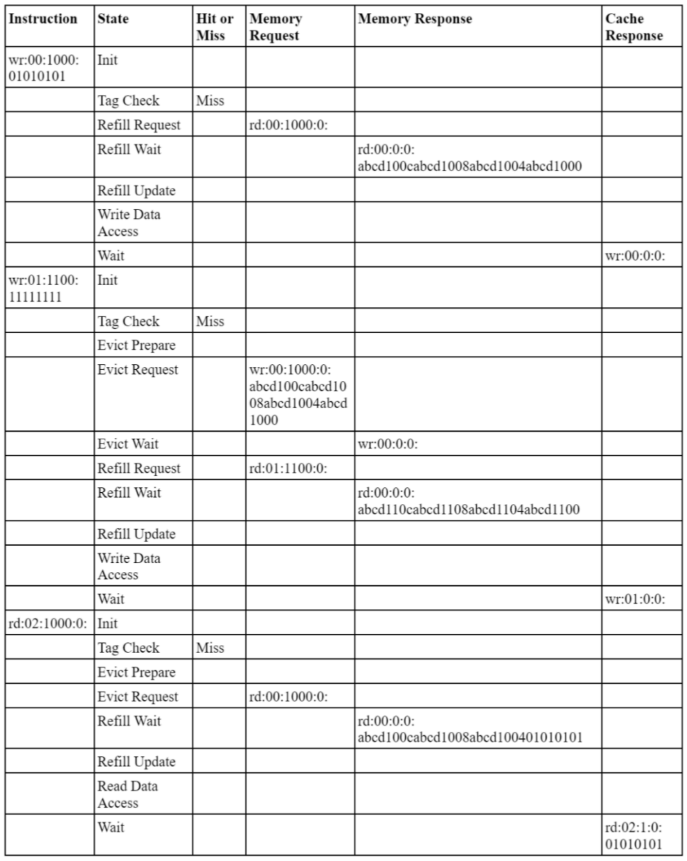
31 8	7 4	3 2	1 0
tag	index	word offset	byte offset

31 7	6 4	3 2	1 0
tag	index	word offset	byte offset

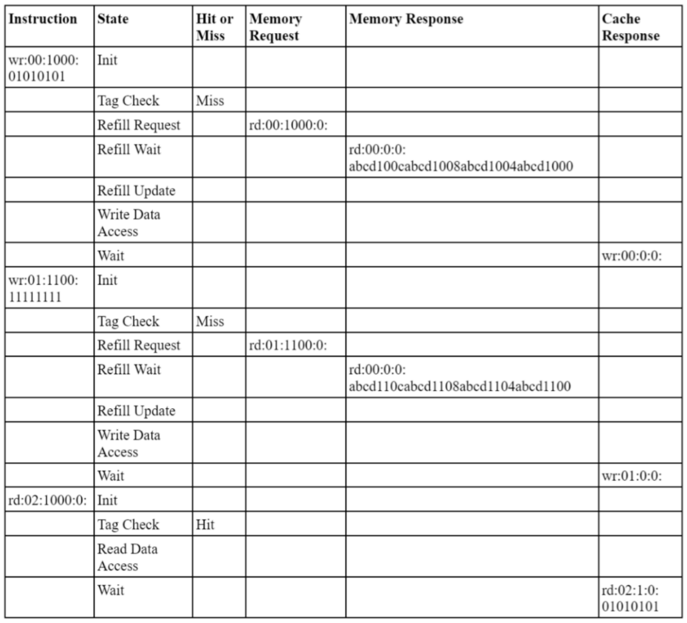
Figure 1: Direct-Mapped Address Map (Left), Two-Way Set-Associative Address Map (Right)

Allowing the data to be placed at one of two locations for a given index will generally reduce the number of conflict misses in a given program. For example, imagine there is a program that accesses data at address 0x1000, then 0x1100, then 0x1000 again. In the direct-mapped implementation, the data access for address 0x1100 would overwrite 0x1000 in the cache, so when it tries to access 0x1000 again, it needs to go to main memory to get it. In the two-way set associative cache, however, no data is evicted from the cache. Address 0x1000 is placed in one way, then 0x1100 is placed in the other way. When the cache tries to access 0x1000 again, it is still in the first way, so it is a cache hit. This sequence of events is shown in *Figure 2* below. Write requests are formatted as wr:opaque:address:data and read requests are formatted as rd:opaque:address:0. Write responses are formatted as wr:opaque:hit(1 indicates hit):len and read responses are formatted as rd:opaque:hit:len:data.

It's also worth noting that even if the program didn't try to access the data at 0x1000 for a second time, the performance of the two-way set associative cache would be better than that of the direct-mapped cache. This is because 0x1000 and 0x1100 have the same index, and the transaction involving 0x1000 is a write. Since that makes it so the data in the cache doesn't match that in main memory, when the cache brings 0x1100 in, it needs to evict 0x1000 and write its new data back to main memory. By contrast, the two-way set-associative cache only needs to write the data for 0x1100 into the second way; it doesn't have to do anything with the data at 0x1000. This can also be seen in *Figure 2* below. The direct-mapped cache evicts then refills the cache, while the two-way set associative cache only refills the cache.



Instruction	State	Hit or Miss	Memory Request	Memory Response	Cache Response
wr:00:1000: 01010101	Init				
	Tag Check	Miss			
	Refill Request		rd:00:1000:0:		
	Refill Wait			rd:00:0:0: abcd100cabcd1008abcd1004abcd1000	
	Refill Update				
	Write Data Access				
	Wait				wr:00:0:0:
wr:01:1100: 11111111	Init				
	Tag Check	Miss			
	Evict Prepare				
	Evict Request		wr:00:1000:0: abcd100cabcd1008abcd1004abcd1000		
	Evict Wait			wr:00:0:0:	
	Refill Request		rd:01:1100:0:		
	Refill Wait			rd:00:0:0: abcd110cabcd1108abcd1104abcd1100	
	Refill Update				
	Write Data Access				
	Wait				wr:01:0:0:
rd:02:1000:0:	Init				
	Tag Check	Miss			
	Evict Prepare				
	Evict Request		rd:00:1000:0:		
	Refill Wait			rd:00:0:0: abcd100cabcd1008abcd100401010101	
	Refill Update				
	Read Data Access				
	Wait				rd:02:1:0: 01010101



Instruction	State	Hit or Miss	Memory Request	Memory Response	Cache Response
wr:00:1000: 01010101	Init				
	Tag Check	Miss			
	Refill Request		rd:00:1000:0:		
	Refill Wait			rd:00:0:0: abcd100cabcd1008abcd1004abcd1000	
	Refill Update				
	Write Data Access				
	Wait				wr:00:0:0:
wr:01:1100: 11111111	Init				
	Tag Check	Miss			
	Refill Request		rd:01:1100:0:		
	Refill Wait			rd:00:0:0: abcd110cabcd1108abcd1104abcd1100	
	Refill Update				
	Write Data Access				
	Wait				wr:01:0:0:
rd:02:1000:0:	Init				
	Tag Check	Hit			
	Read Data Access				
	Wait				rd:02:1:0: 01010101

Figure 2: Direct-Mapped Line Trace (Left) vs Two-Way Set Associative Line Trace (Right)

The alternative design uses two tag arrays and two valid bit arrays, with 8 entries in each array. We use two of these arrays because we want to read the tags and valid bits from both ways in parallel, which allows us to be in the tag check state for only one cycle instead of two. This design only uses one data and dirty bit array, as these reads don't need to happen in parallel. The cache decides what way it's going to work with before the data array or dirty bit array are read, so by the time these arrays need to be read, it only needs to read one location out of both of the arrays; there is no need to read data or dirty bits from both ways, just the way the cache is working with. Since these arrays hold the data for both ways, they have 16 entries each. The datapath for the baseline design is shown in *Figure 3* below, and the datapath for the alternative design is shown in *Figure 4* below.

The cache decides which way to place the data in by using the LRU (least-recently-used) replacement policy. This policy will replace the cache line that was not accessed recently. For example, if way 0 was accessed (read or written) most recently, the cache line in way 1 will get evicted. We keep track of which way was used most recently for a given index by using a use bits array. The use bit is equal to which way was used most recently, and the cache will evict the data in the other way. This bit is set during the wait stage of the FSM.

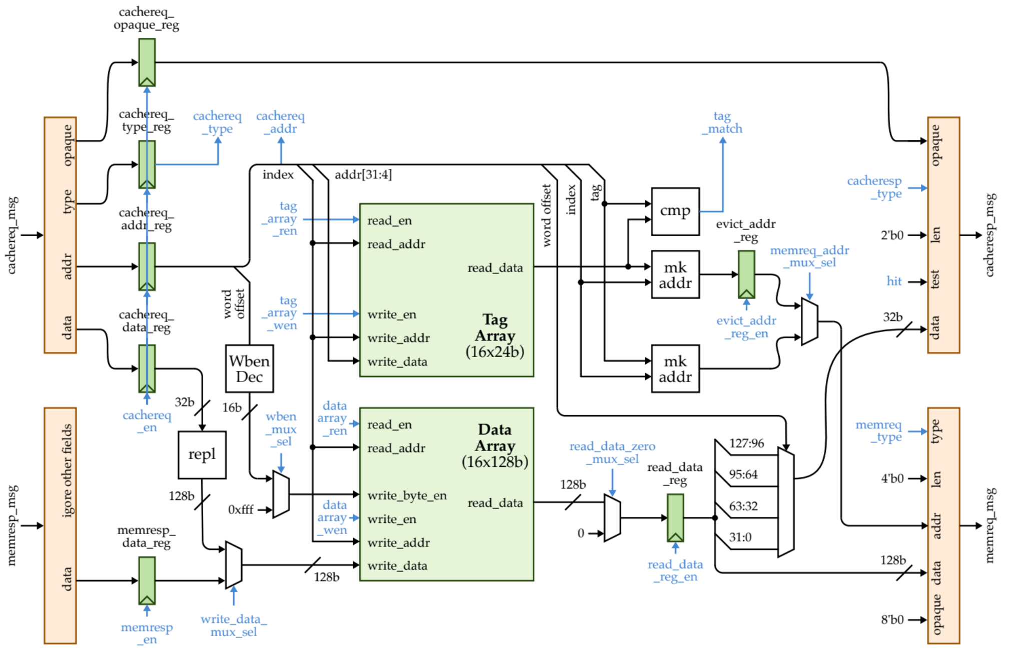


Figure 3: Direct-Mapped Cache Datapath

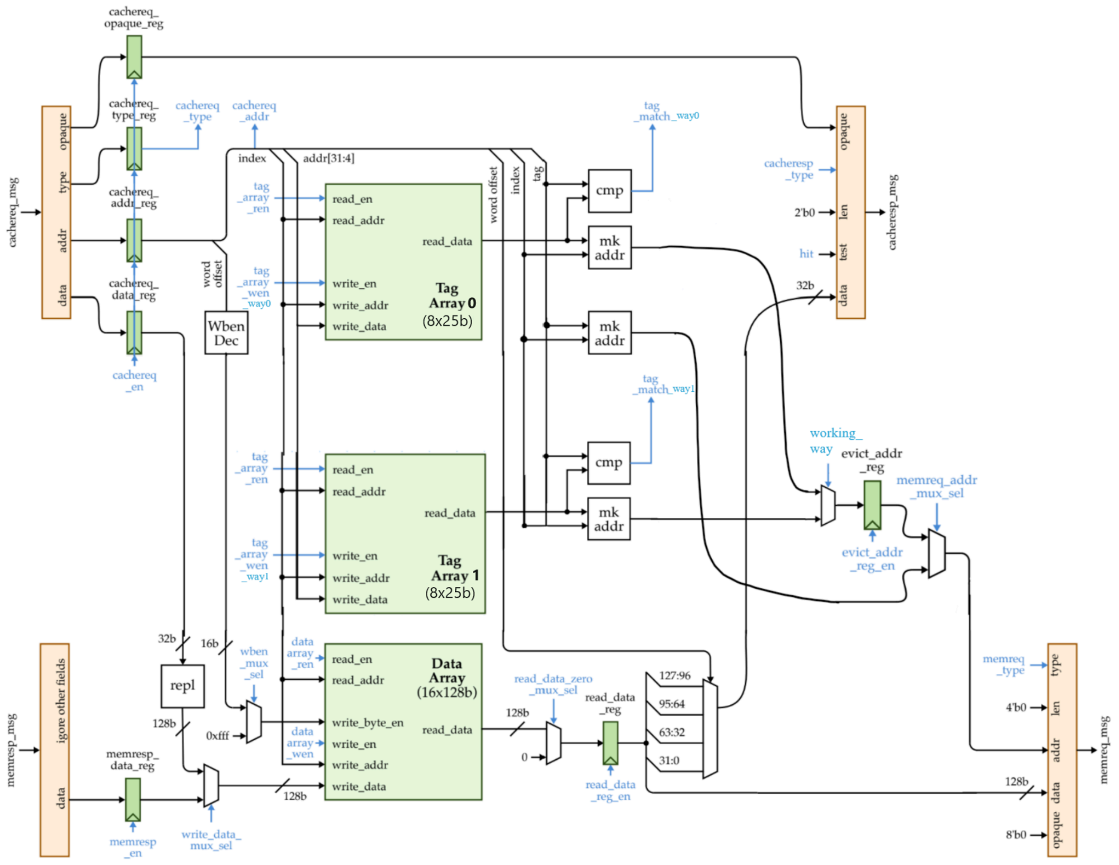


Figure 4: Two-Way Set-Associative Datapath

The control logic between the two implementations is nearly identical. The main difference between them is that the control unit has to check both tag match variables and valid bit arrays to see if it's a hit. If it is a hit, it works with the data in the corresponding way. If it’s a miss, the control unit selects which way to evict and/or overwrite by reading the use bits array at the proper index. The control unit for both implementations is shown below in *Figure 5*.

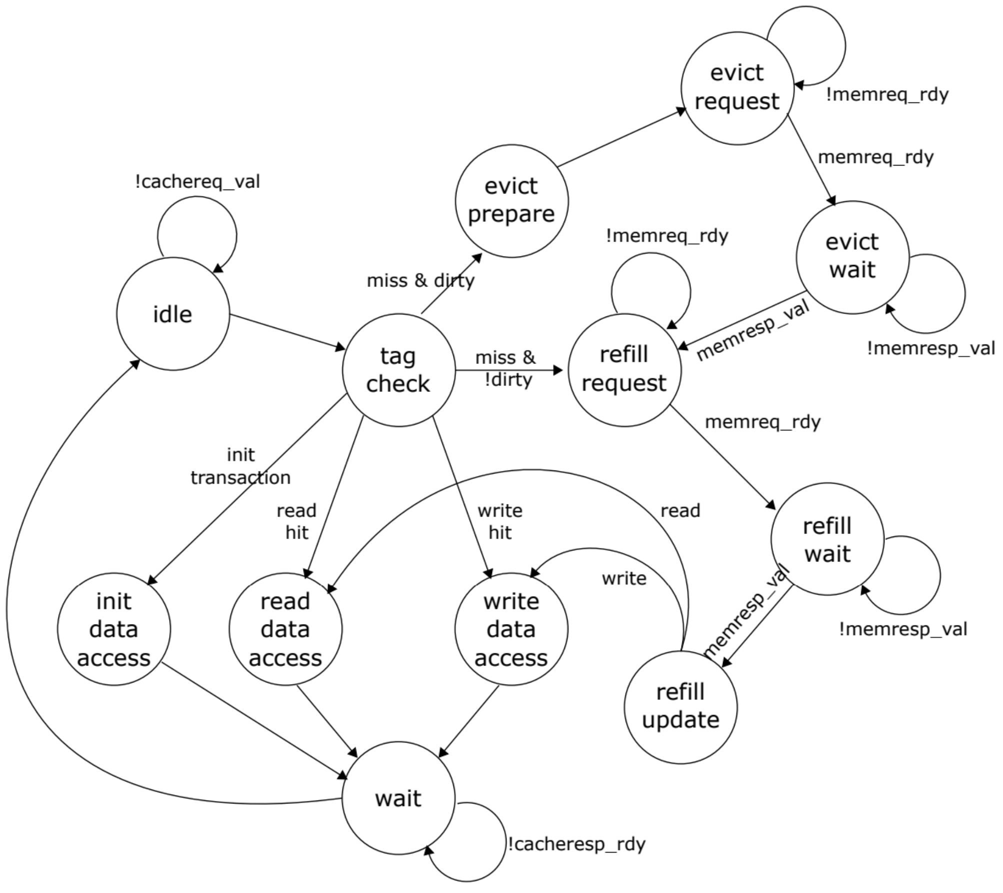


Figure 5: FSM Control Unit

When implementing our designs, we incorporated the patterns and principles of modularity, hierarchy, and encapsulation. Our designs exhibit modularity by decomposing the design into two modules: the datapath and the control unit. Within these modules, we are using smaller modules such as registers, SRAMs, a decoder, tag arrays, data arrays, tag comparators, and muxes. All of these parts are their own module, and they combine to create a much larger, working cache. Everything in the datapath diagram above is its own module, with the exception of the mkaddr block. We decided not to make a module for this since it’s just wiring and not really logic.

Our designs show hierarchy through the decomposition into smaller modules. The CacheAlt.v file is at the highest level, and it ties everything together. Just below that are the datapath and control unit files. Below those are files that include things like the muxes, registers, SRAMs, register files, tag arrays, data arrays, and so on. Through tracing which module calls which other modules, it’s very easy to see what the high level and low level files and blocks are in the hierarchy.

The designs exhibit encapsulation by hiding implementation details from the interface. One such detail is the cache latency. This is hidden using the val/rdy signals. Another detail is which way the data is placed in. The user has no reason to know which way the data is placed in; all they care about is getting the proper data back. When the cache gives a response, it doesn’t include a field for which way the data is stored in the cache.

3. Testing Strategy

We were given a large portion of the directed test cases for this lab. We added directed tests to stress the function of the different implementations and random testing to increase code coverage. Though we were given many tests, we still took an incremental approach to our implementation and testing strategy. We implemented one instruction type at a time, then tested that instruction to ensure functionality before moving on to the next. Because the interface of the implementations is the same, we were able to use the same tests for both. However, we were still able to make tests specific to the direct-mapped and two-way set-associative cache. We implemented our instructions in the order of init, read hit, write hit, read miss, then write miss, and we tested functionality in that order as well.

We began testing our code using unit testing. Unit testing tests specific parts/modules of the design; it doesn't test the design as a whole. This allows us to check that the smaller parts of our design are working properly, which is critical for ensuring that the larger design works. In this lab, we implemented three units that we were able to unit test: the repl unit, the wben decoder, and the cmp unit. We tested each of these modules with simple directed test cases. For each module, we made sure all lines of code were covered with the different test cases. Since these modules are so simple, we only needed about five tests for each module. Once we were confident that these units were working, we moved onto testing the cache as a whole.

We started the full cache tests with directed testing. Directed testing allows us to test specific scenarios; we come up with the requests to send to the cache. This allows us to test specific cases that we think might cause an issue in our cache, or test instances that we think would be corner cases. It also allows us to come up with tests that test very specfic behavior. For example, if I want to test that both ways are being written in the two-way set-associative cache, I would create a test that reads two addresses with the same index but different tags. I would then run the test case, ensure that the outputs look how I expect them to, then examine the line trace. If the cache is going into the evict stage, then there is a problem and only one way is being written. However, if it goes directly into the refill stage, we can be pretty sure that both ways are being used. I can then read from both of those addresses again and look at the line trace. If it indicates a hit for both addresses, then I know the cache is functioning how I expect. The directed test cases that we implemented are shown in *Table 1* below.

After our cache was passing all of the directed test cases, we moved onto random testing. Our random tests generate random data and addresses, then send a request to the cache with that address and data. It then checks that the cache response is correct. This allows us to increase line coverage and test situations that we might not have thought of. It ensures that we're not just testing the cases that pass and ignoring those that don't, and increases the odds that our tests discover a problem if one exists. The random test cases that we implemented are shown in *Table 1* below.

Type	**Category**	**Justification/Description**
Unit	Compare	Test the cmp unit. We test a few different combinations of tags matching and tags not matching. This is a key part to our cache working properly.
	Wben Decoder	Test the wben decoder unit. We test one situation for each of the possible outputs. This is a key part to our cache working properly.
	Repl Unit	Test the repl unit. We test this with a few different data values. This is a key part to our cache working properly.
Directed	Read hit	Test the read hit path. We test this with different tags, indexes, and overall addresses, as well as with different stored data. We test this path both when the dirty bit is set and when the dirty bit is not set. The dirty bit being set should make no difference for this path.
Directed	Write hit	Test the write hit path. We test this with different tags, indexes, and overall addresses, as well as with different data to write. We test this path both when the dirty bit is set and when the dirty bit is not set. The dirty bit being set should make no difference for this path. Both cases need to refill.
Directed	Read miss	Test the read miss path. We test this with different tags, indexes, and overall addresses, as well as with different stored data. We test this path both when the dirty bit is set and when the dirty bit is not set. When the dirty bit is set, the data in the cache needs to be evicted. When the dirty bit is not set, there is no need for eviction. Both cases need to refill. Test for all miss types: conflict, compulsory, and capacity.
Directed	Write miss	Test the read miss path. We test this with different tags, indexes, and overall addresses, as well as with different stored data. We test this path both when the dirty bit is set and when the dirty bit is not set. When the dirty bit is set, the data in the cache needs to be evicted. When the dirty bit is not set, there is no need for eviction. Both cases need to refill. Test for all miss types: conflict, compulsory, and capacity.

Directed	Write data to both ways	Test the set-associative implementation. Write to two addresses that have the same index and a different tag. For the second write, look at the line trace and ensure that the data from the first write is not evicted.
Directed	LRU	Write data to one way, then write to the same index in the other way. Access the data in both ways a couple of times. Write another address with the same index, and examine the line trace to see which cache line was evicted. Ensure it was the line that was used least recently.
Directed	Four-bank	Test our cache for a four-bank cache organization. Ensure that the outputs and line traces are correct.
Random	Read	Generate random addresses. Read the data at that address and ensure that the outputted data is correct.
Random	Read, Write	Randomly decide whether to do a read or a write. Randomly generate an address to work with. If it's a write, randomly generate data to write. Make the cache request and ensure that the response matches what it should be.

Table 1: Test Cases

4. Evaluation

The alternative design has more components and more logic than the baseline design. It has an extra tag array, extra tag comparator, extra valid bit array, extra muxes, and so on. It also has extra logic to set the value of the control and status signals, which it also has more of. All of these extra components and logic require more area. Since the alternative design has more stuff, it will take up more space than the baseline design does.

More components and more logic generally means a longer cycle time, as the data needs to travel a longer distance. However, it is likely that our critical path is actually accessing main memory, not doing the logic before that. Thus, we don't think the critical path will be any different between the baseline and alternative designs. The alternative design likely has the same cycle time as the baseline design.

Powering more components and logic will also take more energy per cycle. In the baseline design, only one tag array and valid bit array are read, and there is no extra logic to decide which way to read or write the data from. The alternative design, however, does have these extra components, all of which require energy to operate. However, since the alternative design will generally take a lot fewer cycles to complete, it will require less overall energy on average.

We can compare the performance between the alternative and baseline designs in terms of the total number of cycles required, the miss rate, and the AMAL (average memory access latency) for different functions.

The loop 1 function (shown in *Table 2*) performs the same across all parameters for the two designs. There is no speedup or slow down for the alternative implementation. This makes sense, as for the direct mapped implementation, addresses 0x1000 through 0x100c will be written to the cache on the first read. Since the data is already in the cache for the next three reads, those will produce a hit. For the next read at 0x1010, the next cache line will be written with the cache line that contains that address, and there will again be three hits and three misses. This pattern of 1 miss then 3 hits repeat until the end of the array is reached. Furthermore, since these are all reads, there is no need to evict and write data to main memory when the same index is eventually reached. The alternative design will have the exact same pattern of 1 miss then 3 hits; it will fill way 0 with the first couple addresses, then fill way 1 with the next ones, then overwrite way 0 with the next ones, and so on. Again, because we are only reading the data, there is no need to evict and send the data in the cache to main memory. All memory requests are due to being the first access for both implementations. This performance is shown in *Figure 6*.

The loop 2 function (shown in *Table 2*) shows better performance for the alternative design in all three categories. This is because accessing array a and array b will always produce the same index, as we're iterating through them at the same time and at the same rate, and they started with the same index. Because of this, the baseline design is forced to bring in the cache line corresponding to array a, read it, then immediately refill that index with the cache line corresponding to array b. In contrast, the alternative design will store the cache line corresponding to array a in way 0 and the cache line corresponding to array b in way 1. It isn't forced to read the a array then immediately overwrite it. Because of this, the alternative design has ¼ of the misses and nearly ⅓ of the number of cycles than that of the baseline design. It's able to cut down the AMAL by 276%. This performance difference is shown in *Figure 6*.

Loop 3 (shown in *Table 2*) shows better performance for the baseline design. This is because in the baseline design, the index of address 0x3080 is 8, while in the alternative design, its index is 0. The indexes of the other arrays are 0 in both of the implementations. In the direct-mapped cache, index 0 is written with a, then overwritten with b, then index 8 is written with c. The next time around, index 0 is written with a, then overwritten with b, and c is already in the cache. The second time around will be repeated until the address maps to the next index. Thus, for a given cache line, the baseline design will have 3 misses the first time around, then 2 misses the next 3 times, making a total of 9 misses. In the alternative design, way 0 is written with a, then way 1 is written with b, then way 0 is written with c. The second time around, way 1 is written with a, way 0 is written with b, and way 1 is written with c. We get into a nasty loop where there can be no hits, as the data that is needed is overwritten in the previous cycle. Because of this, the alternative design has 1.3 times the misses and 1.27 times the number of cycles. Its AMAL increases by 127%. This performance difference is shown in *Figure 6*.

Loop 4 (shown in *Table 2*) shows better performance for the alternative design. This makes sense, as the array locations mean that both arrays will always have the same index as we iterate through them. The alternative design will always have array a in one way, and array b in the other way. It will not need to overwrite itself when working through the same cache line. The baseline design, however, will access array a twice, then overwrite it with array b. Then, in the next cycle, it will need to overwrite array b with array a. It is clear that this will result in more misses for the baseline design. This is reflected in the performance metrics. We can see that the alternative design has 2/9 of the misses and roughly ⅓ the number of cycles. Its AMAL is 302% smaller than that of the baseline design. These performance differences are shown in *Figure 6*.

Loop 5 (shown in *Table 2*) also shows better performance for the alternative design. Every single array has the same index, and the baseline design will be forced to overwrite itself more often than the alternative design, as it can only place data in one location, while the alternative design can place data in two locations. As a result, the alternative design has 0.9 times the number of cycles, ⅘ the number of misses, and its AMAL is 113% smaller than the baseline design. These performance differences are shown in *Figure 6*.

It's clear to see that there are tradeoffs between performance and cycle time, area, and energy per cycle. We would choose our alternative design going forward because of the better performance in most situations. Although it takes more energy per cycle, we see that the total number of cycles is frequently a lot less, so the total energy for the alternative design will generally be smaller. We believe that even though this will require more area and possibly a longer cycle time, it will be worth it to cut down on total energy consumption and to increase performance.

// **loop 1 pattern**
// a array allocated at 0x1000
for (i = 0; i < 100; i++)
    result += a[i];
	// **loop 2 pattern**
	// a array allocated at 0x1000
	// b array allocated at 0x2000
	for (i = 0; i < 100; i++)
	    result += a[i] * b[i];
		// **loop 3 pattern**
		// a array allocated at 0x1000
		// b array allocated at 0x2000
		// c array allocated at 0x3080
		for (i = 0; i < 100; i++)
		    result += a[i] * b[i] + c[i];

// **loop 4 pattern**
// a array allocated at 0x1000
// b array allocated at 0x2000
for (i = 0; i < 100; i++)
    a[i] = a[i] + b[i];
	// **loop 5 pattern**
	// a array allocated at 0x1000
	// b array allocated at 0x2000
	// c array allocated at 0x3000
	// d array allocated at 0x4000
	for (i = 0; i < 100; i++)
	    a[i] = a[i] + b[i];
	    c[i] = a[i] + d[i];	

Table 2: Loops Used for Evaluations

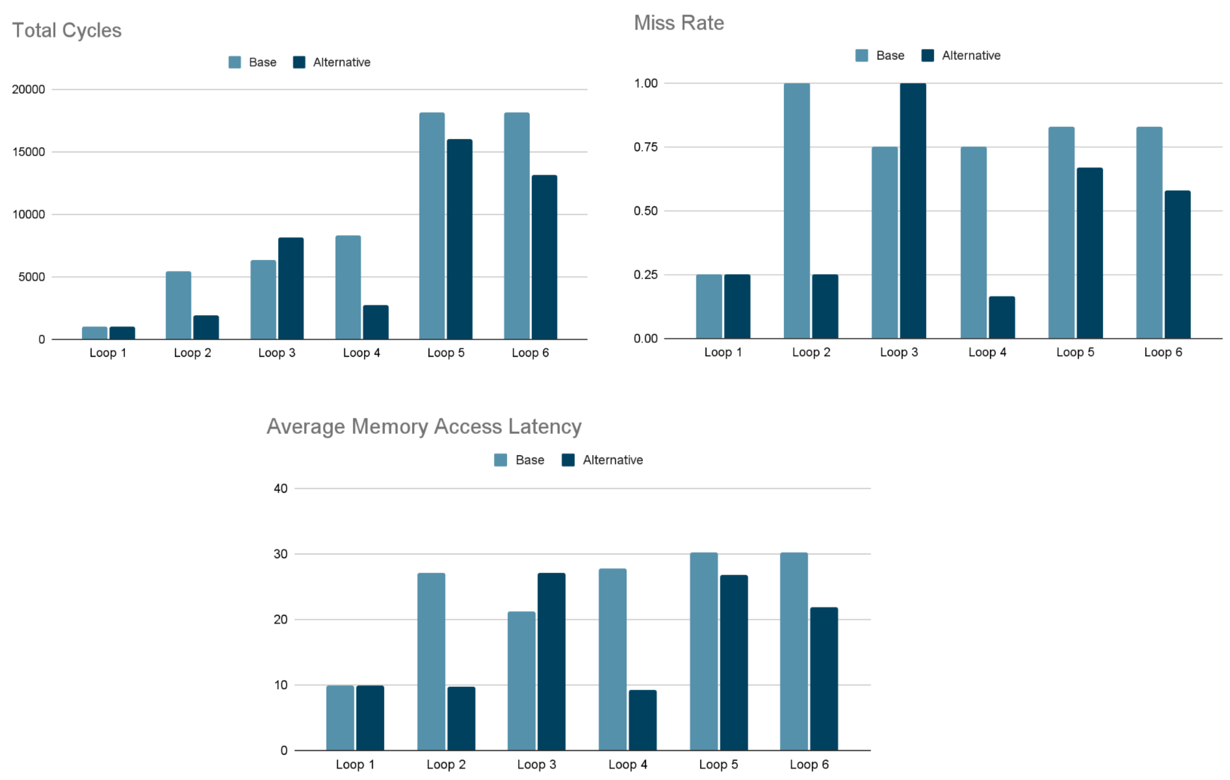


Figure 6: Performance Comparison Between Direct-Mapped and 2-Way Set-Associative Caches

5. Conclusion

The alternative design performs better than the baseline design for most functions, especially those that iterate over more than one array. The alternative design will generally reduce the number of conflict misses, and even though it has a bigger area, its total energy consumption will be smaller due to needing fewer cycles to complete. If the machine will usually be working with one array at a time, a direct-mapped cache is the better choice due to the increased area, complexity and energy per cycle that the alternative design requires. Loop 1 saw the same performance between the alternative and baseline designs, which makes sense given what the loop was accessing. Loop 2, 4, and 5 resulted in the alternative design having better performance, with the AMAL being only 36%, 33%, and 88% of the AMAL of the baseline design, respectively. For loop 3, the alternative design performed worse, with an amal 127% that of the baseline design. However, loop 3 could have been optimized by having two loops: one loop that multiplied a[i] and b[i], then another loop that added c[i]. This would've gotten rid of the nasty problem of the data being overwritten right before we wanted to use it, forcing the cache to access main memory. If we have a compiler that can do these optimizations or a programmer that codes with this in mind, the clear decision going forward is the alternative design. Even without that, it would still make the most sense to use the alternative design in the future, as it has better performance for most loops. The alternative design generally has a better AMAL, total number of cycles, and miss rate.

6. Work Distribution

Annie began the baseline datapath and control path. Ashley completed the baseline implementation, unit tests, and the alternative design and implementation. Annie created the additional test cases and loops for evaluation. The report was written together.