CS152 Computer Architecture and Engineering

Caches and the Memory Hierarchy

SOLUTION

Assigned *02/08/2024*	Problem Set #2, Version (1.0)	*Due February 21* *@ 11:59:59PT*

http://inst.eecs.berkeley.edu/~cs152/sp24

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solution to the problems.

The problem sets also provide essential background material for the exam and the midterms. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets yourself you are unlikely to succeed on the exam or midterms!

By grading primarily on an effort basis, we mean that we will award significant partial credit for demonstrating your understanding of the problem and concepts at hand. As long as reasonable assumptions and explanations are provided, we will lean towards awarding credit.

We will distribute solutions to the problem set after the deadline to give you feedback.

Assignments must be submitted through Gradescope by **11:59:59pm PT** on the specified due date. *Box/clearly mark all solutions that don't involve filling in a figure/table. Only boxed/clearly marked solutions and filled in figures/tables will be considered for grading.* See the course website for the policy on slip days (late submissions).

Name: ____

SID: ____

Collaborators (Name, SID):

____

Problem 1: Cache Access-Time & Performance

This problem requires the knowledge of Handout #2 and the Lectures on Memory. Please, read these materials before answering the following questions.

Jessie is trying to determine the best cache configuration for a new processor. She knows how to build two kinds of caches: direct-mapped caches and 4-way set-associative caches. The goal is to find the better cache configuration with the given building blocks. She wants to know how these two different configurations affect the **clock speed** and the **cache miss-rate** and choose the one that provides <u>better performance in terms of average latency</u> for a load.

Problem 1.A Access Time: Direct-Mapped

First, we want to compute the access time of a direct-mapped cache. We use the implementation shown in Figure H2-A in Handout #2. Assume a 256-KB (kibibyte = 2^{10} bytes) cache with 8-word (32-byte) cache lines. The address is 32 bits and byte-addressed, so the two least significant bits of the address are ignored since a cache access is word-aligned. The data output is also 32 bits (1 word), and the MUX selects one word out of the eight words in a cache line. Using the delay equations given in Table 2.1-1, **fill in the column for the direct-mapped (DM) cache in the table**. Use the ceiling of the logarithm to get an integer, if needed. *In the equation for the data output driver, 'associativity' refers to the associativity of the cache (1 for direct-mapped caches, A for A-way set-associative caches).*

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	30×(# of index bits) + 80	Tag	470	410
		Data	470	410
Memory array	$30\times \log_2$(# of rows) + $30\times \lceil \log_2$(# of bits in a row)$\rceil$ + 100	Tag	610	640
		Data	730	730
Comparator	30×(# of tag bits) + 70		490	550
N-to-1 MUX	$50\times\log_2 N + 100$		250	250
Buffer driver	180			180
Data output driver	50×(associativity) + 100		150	300
Valid output driver	40		40	40

Table 2.1-1: Delay of each Cache Component

i) **What is the critical path of this direct-mapped cache for a cache read?**

ii) **What is the access time of the cache (the delay of the critical path)? To compute the access time, assume that a 2-input gate (AND, OR) delay is 50 ps.**

iii) **If the CPU clock is 2.5 GHz, how many CPU cycles does a cache access take?**

For the given cache structure which is byte addressable, we can know that the **# of offset bits = log$_2$(# of byte in a word line) = 5 bits.**

We know the **# of lines = $ size / wordline size = $2^{18}/2^5 = 2^{13}$ lines**

Because the cache is direct map, then the **# of index bit = log$_2$(2^{13}) = 13 bits**

Because the total address bits is 32 bits, then **# of tag bits = 32-13-5 = 14 bits**

Applying all values we calculate above to the delay equations, we have:

Decoder (tag) = 30 * 13 + 80 = 470ps

Decoder (data) = 30 * 13 + 80 = 470ps

Note: # of bits in a row for the tag should include the valid and dirty bits

Memory array (tag) = 30*log$_2$(2^{13}) + 30*ceil(log$_2$(14+2)) + 100 = 610ps

Memory array (data) = 30*log$_2$(2^{13}) + 30*ceil(log$_2$(32*8)) + 100 = 730ps

Comparator = 30*14+70 = 490ps
N-1 mux = 50*log$_2$(8) + 100 = 250ps
Data output driver = 50 * 1 + 100 = 150ps

To determine the critical path for a cache read, we need to compute the time it takes to go through each path in hardware (tag check and data read). By taking the maximum delay of these two paths, we are left with the critical path.

Time to tag check valid driver from tag array
= Decoder (tag) + Memory array (tag) + comparator + AND gate + valid output driver
= 470 + 610 + 490 + 50 + 40 = 1660ps

Time to data output drive from data array
= Decoder (data) + Memory array (data) + 8-1 MUX + data output driver = 470 + 730 + 250 + 150 = 1600ps

From the above results, we can see that the critical path is tag check. **The access time is 1660ps**. At 2.5GHz, the cache access takes **(1660ps/(1/2.5GHz)) = 4.15 ~ 5 cycles.** Here, **rounding up** to the nearest cycle is sensible, as this reflects how a synchronous system would work.

Problem 1.B Access Time: Set-Associative

We also want to investigate the access time of a set-associative cache using the 4-way set-associative cache in Figure H2-B in Handout #2. Assume the total cache size is still 256-KB (each way is 64KB), a 2-input gate delay is 50 ps, a 4-input gate delay is 100 ps, and all other parameters (such as the input address, cache line, etc.) are the same as part 2.1.A. **Compute the delay of each component and fill in the column for a 4-way set-associative cache in Table 2.1-1.**

i) What is the critical path of the 4-way set-associative cache?

ii) What is the access time of the cache (the delay of the critical path)?

iii) What is the main reason that the 4-way set-associative cache is slower than the direct-mapped cache?

iv) If the CPU clock is 2.5 GHz, how many CPU cycles does a cache access take?

For the given cache structure which is byte addressable, we know that the **# of offset bits = $\log_2$(# of byte in a word line) = 5 bits.**

We know that the **# of lines = ($ size / wordline size) / nWays = $(2^{18}/2^{5}) / 4 = 2^{11}$ lines**

The number of index bits is then **# of index bit = $\log_2(2^{11}) = 11$ bits**

The total address bits is 32 bits, then the **# of tag bits = 32-11-5 = 16 bits**

Applying all values we calculate above to the delay equations, we have:

Decoder (tag) = 30 * 11 + 80 = 410ps

Decoder (data) = 30 * 11 + 80 = 410ps

Note: tag bits include the valid/dirty bits (+2)
Memory array (tag) = $30*\log_2(2^{11}) + 30*\text{ceil}(\log_2((16+2)*4)) + 100 = 640$ps

Memory array (data) = $30*\log_2(2^{11}) + 30*\text{ceil}(\log_2(32*8*4)) + 100 = 730$ps

Comparator = 30*16+70 = 550ps
N-1 mux = $50*\log_2(8) + 100 = 250$ps
Data output driver = 50 * 4 + 100 = 300ps

There are three possible critical paths in an associative cache. The first two are the same as those in the direct mapped cache. The third one is the path through the tag array, the tag comparators, through the way-select mux, and through the data output driver.

Time to tag check valid driver
= Decoder (tag) + Memory array (tag) + comparator + AND gate + OR gate + valid output

driver
= 410 + 640 + 550 + 50 + 100 + 40 = 1790

Time to data output drive:
= Decoder (data) + Memory array (data) + 8-1 MUX + data output driver = 410 + 730 + 250 + 300 = 1690ps

Time to tag valid check to output driver:
= Decoder (tag) + Memory array (tag) + comparator + AND gate + buffer driver + data output driver
= 410 + 640 + 550 + 50 + 180 + 300 = 2130ps

From the above results, we can see that the critical path is tag valid check to output driver. **The access time is 2130ps**. At 2.5GHz, the cache access takes **(2130ps/(1/2.5GHz)) = 5.3 ~ 6 cycles.** Here, **rounding up** to the nearest cycle is sensible, as this reflects how a synchronous system would work.

Problem 1.C — Miss-rate analysis

Now Ben is studying the effect of set-associativity on the cache performance. Since he now knows the access time of each configuration, he wants to know the miss-rate of each one. For the miss-rate analysis, Ben is considering two small caches: a direct-mapped cache with 8 lines with 32 bytes/line, and a 4-way set-associative cache of the same size and line size. For the set-associative cache, Ben tries out two replacement policies – least recently used (LRU) and round robin (FIFO).

Ben tests the cache by accessing the following sequence of hexadecimal byte addresses, starting with empty caches. For simplicity, assume that the ***addresses are only 12 bits***. **Complete the following tables by filling in the hexadecimal tag values** for the direct-mapped cache and both types of 4-way set-associative caches showing the progression of cache contents as accesses occur (in the tables, 'inv' = invalid, and the column of a particular cache line contains the tag of that line). Also, for each address calculate the tag and index (which should help in filling out the table). *You only need to fill in elements in the table when a value changes.*

Address: 12 bits
Tag: 4 bits [11:8]
Index: 3 bits [7:5]
Offset: 5 bits [4:0]

Address in Binary

D-map	**Addresses and tags are in HEX**								
	line in cache (tag)								hit?
Address	L0	L1	L2	L3	L4	L5	L6	L7	
11B	1	inv	inv	inv	inv	inv	inv	inv	no
134		1							no
20D	2								no
1A2						1			no
105	1								no
360				3					no
27D				2					no
121		1							yes
1A3						1			yes
17A				1					no
307	3								no
273				2					no
131		1							yes

	Direct-Mapped
Total Misses	10
Total Accesses	13

Address: 12 bits
Tag: 6 bits [11:6]
Index: 1 bits [5:5]
Offset: 5 bits [4:0]

Address in Binary

4-way	**LRU -- addresses and tags are in HEX**								
	line in cache								hit?
Address	Set 0				Set 1				
	way0	**way1**	**Way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
11B	4	inv	inv	inv	inv	inv	inv	inv	no
134					4				no
20D		8							no
1A2						6			no
105	-								yes
360							D		no
27D								9	no
121					-				yes
1A3						-			yes
17A							5		no
307			C						no
273								-	yes
131					-				yes

	4-way LRU
Total Misses	8
Total Accesses	13

Address in Binary

4-way	**FIFO -- addresses and tags are in HEX**								
	line in cache (tag)								hit?
Address	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
11B	4	inv	inv	inv	inv	inv	inv	inv	no
134					4				no
20D		8							no
1A2						6			no
105	-								yes
360							D		no
27D								9	no
121					-				yes
1A3						-			yes
17A					5				no
307			C						no
273								-	yes
131						4			no

	4-way FIFO
Total Misses	9
Total Accesses	13

Problem 1.D Average Latency

Assume that the results of the above analysis can represent the average miss-rates of the direct-mapped and the 4-way set-associative 256-KB caches studied in 1.A and 1.B.

i) What would be the average memory access latency in CPU cycles for each cache? Assume that the cache miss penalty is 20 cycles and use cache access cycle count from 1.A and 1.B. Which one is better?

ii) For the different replacement policies for the set-associative cache, which one has a smaller cache miss rate for the address stream in 1.C? Explain why.

iii) Is that replacement policy always going to yield better miss rates? If not, give a counter example using an address stream.

The miss rate for the direct-mapped cache is 10/13. The miss rate for the 4-way LRU set associative cache is 8/13. For FIFO is 9/13.

The average memory access latency is **(hit time) + (miss rate) × (miss penalty)**.

For the direct-mapped cache, the average memory access latency would be:
(5 cycles) + (10/13) × (20 cycles) = 20.4 cycles.
For the LRU set-associative cache, the average memory access latency would be: (6 cycles) + (8/13) × (20 cycles) = 18.3 cycles.
For the FIFO set-associative cache, the average memory access latency would be: (6 cycles) + (9/13) × (20 cycles) = 19.8 cycles.

The set-associative cache with LRU replacement is better than the direct-mapped cache in terms of average memory access latency.
For the above example, LRU has a slightly smaller miss rate than FIFO. This is because the FIFO policy replaced tag{4} block instead of tag {D} during the 10th access, because the {4} block has been in the cache longer, even though the {D} was least recently used. In this case, **the LRU policy took better advantage of temporal locality.**

LRU does not always outperform FIFO. Assume we have a set-associative cache with the same parameters as in 1.C and an access sequence shown below. There is a miss with LRU for the last access while there is a hit with FIFO.

```
0x11B
0x134
0x20D
0x1A2
0x105
0x360
0x27D
0x121
0x1A3
0x17A
```

```
0x307
0x273
0x361
```

Problem 2: Loop Ordering

This problem requires knowledge of Lecture 7. Please, read it before answering the following questions.

This problem evaluates the cache performances for different loop orderings. You are asked to consider the following two loops, written in C, which calculate the sum of the entries in a 128 by 32 matrix of 32-bit integers:

Loop A	***Loop B***

`sum = 0;`
`for (i = 0; i < 128; i++)`
`  for (j = 0; j < 32; j++)`
`    sum += A[i][j];`
	`sum = 0;`
	`for (j = 0; j < 32; j++)`
	`  for (i = 0; i < 128; i++)`
	`    sum += A[i][j];`

The matrix `A` is stored contiguously in memory in row-major order. Row major order means that elements in the same row of the matrix are adjacent in memory as shown in the following memory layout:

`A[i][j]` resides in memory location `[4*(32*i + j)]`

Memory Location:

0	4		124	128		4*(32*127+31)
A[0][0]	A[0][1]	...	A[0][31]	A[1][0]	...	A[127][31]

For *Problem 2.A* to *Problem 2.C*, assume that the caches are initially empty. Also, assume that only accesses to matrix `A` cause memory references and all other necessary variables are stored in registers. Instructions are in a separate instruction cache.

Problem 2.A

Consider an 8KB direct-mapped data cache with 4-word (16-byte) cache lines.
Calculate the number of cache misses that will occur when running Loop A.
Calculate the number of cache misses that will occur when running Loop B.

Each element of the 128x32 matrix A can only be mapped to one particular cache location in this direct-mapped data cache. Since each row has 32 32-bit integers, and since each cache line can hold 4 32-bit ints, a row of the matrix occupies the lines in 8 consecutive sets of the cache.

Loop A—where each iteration of the inner loop sums a row of A—accesses memory addresses in a linear sequence. Given this access pattern, the access to the first word in each cache line will miss, but the next three accesses will hit. After sequentially moving through this line, it will not be accessed again, so its later eviction will not cause any future misses. Therefore, Loop A will only have compulsory misses for the 1024 (128 rows x 8 lines per row) first-word-in-line accesses that matrix A spans.

The consecutive accesses in Loop B will move in a stride of 32 words. Therefore, the inner loop will touch the first element in 128 cache lines before the next iteration of the outer loop. While intuition might suggest that the 128 lines could all fit in the cache with 512 sets, there is a complicating factor: each row is eight cache lines past the previous row, meaning that the lines accessed when traversing the first column go in indices 0, 8, 16, 32, and so on. Since the lines containing the column are competing for only one eighth of the total number of sets (effectively 64 sets), the lines loaded when starting a column are evicted by the time the column is complete, preventing any reuse. Therefore, all 4096 (128 x 32) accesses miss.

The number of cache misses for Loop A:____________ 1024______________

The number of cache misses for Loop B:____________ 4096 _____________

Problem 2.B

Consider a direct-mapped data cache with 4-word (16-byte) cache lines.
Calculate the minimum number of cache lines required for the data cache if Loop A is to run without any cache misses other than compulsory misses.
Calculate the minimum number of cache lines required for the data cache if Loop B is to run without any cache misses other than compulsory misses.

Since Loop A accesses memory sequentially, we can sum all the elements in a cache line and then never touch it again. Therefore, we only need to hold 1 active line at any given time to avoid all but compulsory misses.

For Loop B to run without any cache misses other than compulsory misses, the data cache needs to have the ability to hold one column of matrix A in the cache. Since the consecutive accesses in the inner loop of Loop B will use one out of every eight cache lines, and since we have 128 rows, Loop B requires 1024 (128 × 8) lines to avoid all but compulsory misses.

Data-cache size required for Loop A: _______________1 ______________ cache line(s)

Data-cache size required for Loop B: _______________1024 ____________ cache line(s)

Problem 2.C

Consider a 8KB set-associative data cache with 4 ways, and 4-word (16-byte) cache lines. This data cache uses a first-in/first-out (FIFO) replacement policy.
Calculate the number of cache misses that will occur when running Loop A.
Calculate the number of cache misses that will occur when running Loop B.

Note that the offset is 4 bits.
The # of lines in a way of this cache = 2^13 / (2^4 * 4) = 2^7=128.

Loop A still only has 1024 (128 rows x 8 lines per row) compulsory misses.

Loop B still cannot fully utilize the cache. Consider accessing a single column. The first 128/8=16 accesses will allocate into way 1 in sets 0, 8, 16, 32, etc.; the next 16 accesses will allocate into way 2 of those same sets; and so on. After 64 accesses, all four ways will be filled, and the next 16 accesses along the column will evict the previous lines in way 1, preventing any reuse. Therefore, all 4096 (128 x 32) accesses miss.

The number of cache misses for Loop A:_____________ 1024 _______________

The number of cache misses for Loop B:_____________ 4096 _______________

Problem 3: Microtagged Cache

In this problem, we explore *microtagging*, a technique to reduce the access time of set-associative caches. Recall that for associative caches, the tag check must be completed before load results are returned to the CPU, because the result of the tag check determines which cache way is selected. Consequently, the tag check is often on the critical path.

The time to perform the tag check (and, thus, way selection) is determined in large part by the size of the tag. We can speed up way selection by checking only a subset of the tag—called a microtag—and using the results of this comparison to select the appropriate cache way. Of course, the full tag check must also occur to determine if the cache access is a hit or a miss, but this comparison proceeds in parallel with way selection. We store the full tags separately from the microtag array.

We will consider the impact of microtagging on a 4-way set-associative 16KB data cache with 32-byte lines. Addresses are 32 bits long. Microtags are 8 bits long. The baseline cache (i.e. without microtagging) is depicted in Figure H2-B in Handout #2. Figure 1, below, shows the modified tag comparison and driver hardware in the microtagged cache.

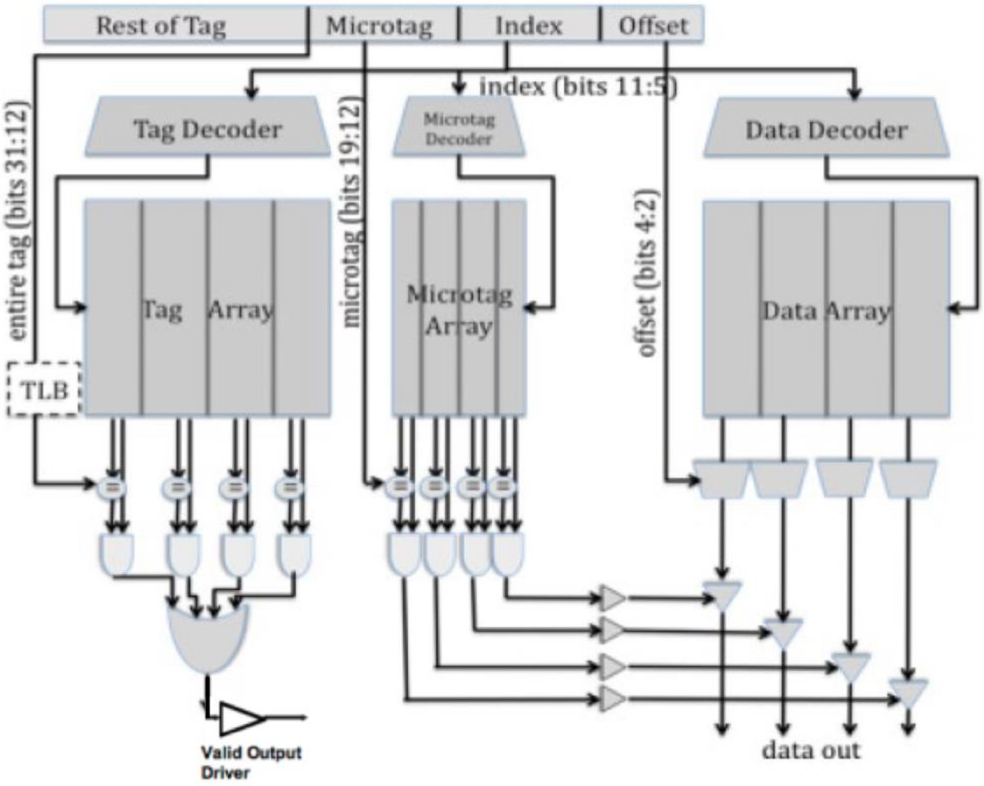


Figure 2.4-1: Microtagged cache datapath

Problem 3.A (PRACTICE - OPTIONAL) Cache Cycle Time

Table 2.4-1, below, contains the delays of the components within the 4-way set-associative cache, for both the baseline and the microtagged cache. For both configurations, determine the critical path and the cache access time (i.e., the delay through the critical path).

Assume that the 2-input AND gates have a 50ps delay and the 4-input OR gate has a 100ps delay.

Component	Delay equation (ps)		Baseline	Microtagged
Decoder	20×(# of index bits) + 100	Tag	240	240
		Data	240	240
		Microtag		240
Memory array	20×$\log_2$ (# of rows) + 20×$\log_2$ [(# of bits in a row)] + 100	Tag	380	380
		Data	440	440
		Microtag		340
Comparator	20×(# of tag bits) + 100	Tag	500	500
		Microtag		260
N-to-1 MUX	50×$\log_2$ N + 100		250	250
Buffer driver	200		200	200
Data output driver	50×(associativity) + 100		300	300
Valid output driver	100		100	100

Table 2.4-1: Delay of each Cache Component

i) What is the old critical path? The old cycle time (in ps)?

Candidate 1: Full tag check
tag decoder → tag read → comparator → 2-in AND → 4-in OR → valid output driver
240 ps + 380 ps + 500 ps + 50 ps + 100 ps + 100 ps = 1370 ps

Candidate 2: Data select based on full tag check
tag decoder → tag read → comparator → 2-in AND → buffer driver → data output driver
240 ps + 380 ps + 500 ps + 50 ps + 200 ps + 300 ps = 1670 ps

Candidate 3: Data readout
data decoder → data read → 4-to-1 MUX → data output driver
240 ps + 440 ps + 250 ps + 300 ps = 1230ps

The critical path is the data select based on the full tag match. The cycle time is 1670 ps.

ii) What is the new critical path? The new cycle time (in ps)?

Candidate 1: Full tag check
same as baseline full tag check => 1370 ps

Candidate 2: Data select based on microtag check
μtag decoder → μtag read → comparator → 2-in AND → buffer driver → data out driver
240 ps + 340 ps + 260 ps + 50 ps + 200 ps + 300 ps = 1390 ps

Candidate 3: Data readout
same as baseline data read => 1230 ps
The critical path is the data select based on the microtag check. The cycle time is 1390 ps.

Problem 3.B AMAT

Assume temporarily that both the baseline cache and the microtagged cache have the same hit rate, 90%, and the same average miss penalty, 15 ns. Using the cycle times 1.5 ns and 1.2 ns for the baseline and microtag caches respectively, compute the average memory access time for both caches.

i) What was the old baseline AMAT (in ns)?

ii) What is the new AMAT (in ns)?

AMAT = (hit_time) + (miss)_rate x (miss_penalty)
= X + (0.1) * (15ns) = X + 1.5ns, where X is the hit time

Old AMAT = 1.5 + 1.5 = 3 ns
New AMAT with microtags = 1.2 + 1.5 = 1.7 ns

Problem 3.C Constraints

Microtags add an additional constraint to the cache: in a given cache set, all microtags must be unique. This constraint is necessary to avoid multiple microtag matches in the same set, which would prevent the cache from selecting the correct way.

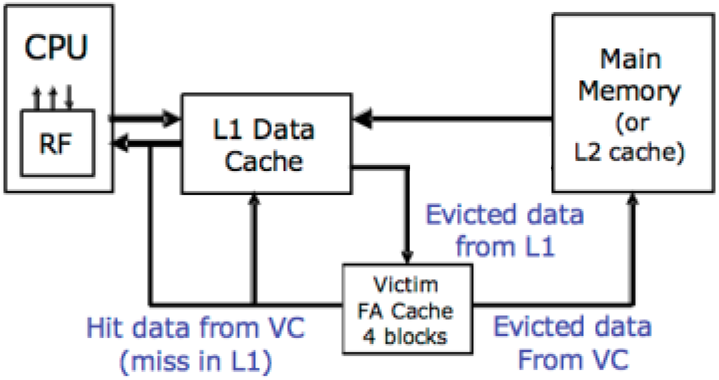
i) State which of the 3C's of cache misses this constraint affects.
ii) How will the cache miss rate compare to an ordinary 4-way set-associative cache?
iii) How will it compare to that of a direct-mapped cache of the same size?
iv) Which 8 bits of the tag might you want to use for the microtag and why?

Because the uniqueness property of microtags restricts the replacement policy, the cache isn't free to make as optimal replacement decisions as it could in the baseline. This will lead to some increase in conflict misses. The magnitude of this effect depends on which 8 bits are selected to form the microtag. In principle, using the bottom 8 bits would result in more potential for microtag collisions and would add the biggest restriction to the ability of the cache to hold spatially local data – data within 2^{12} to 2^{20} bytes of each other. The same argument could be used for choosing the top 8 bits. When addressing data that is spatially local, it will likely have the same upper tag bits. But due to our constraint of uniqueness, this would also cause many cache conflicts. Thus, we may want to choose 8 bits somewhere in the middle of the tag, depending on our application. Regardless, the microtagged cache will still be better than a direct mapped cache of the same size and line size.

Problem 4: Victim Cache Evaluation

Although direct-mapped caches have an advantage of smaller access time than set- associative caches, they have more conflict misses due to their lack of associativity. In order to reduce these conflict misses, Norm Jouppi proposed victim caching, where a small fully-associative back up cache, called a victim cache, is added to a direct-mapped L1 cache to hold recently evicted cache lines.

The following diagram shows how a victim cache can be added to a direct-mapped L1 data cache. Upon a data access, the following chain of events takes place:



1. The L1 data cache is checked. If it holds the data requested, the data is returned.
2. If the data is not in the L1 cache, the victim cache is checked. If it holds the data requested, the data is moved into the L1 cache and sent back to the processor. The data evicted from the L1 cache is put in the victim cache, and put at the end of the FIFO replacement queue.
3. If neither of the caches holds the data, it is retrieved from memory, and put in the L1 cache. If the L1 cache needs to evict old data to make space for the new data, the old data is put in the victim cache and placed at the end of the FIFO replacement queue. Any data that needs to be evicted from the victim cache to make space is written back to memory or discarded, if unmodified.

Note that the two caches are *exclusive*. That means that the same data cannot be stored in both L1 and victim caches at the same time.

Problem 4.A (PRACTICE - OPTIONAL) Baseline Cache Design

The diagram below shows our victim cache, a 32-byte fully associative cache with four 8-byte cache lines. Each line contains two 4-byte words and has an associated tag and two status bits (valid and dirty). The Input Address is 32-bits. Since the cache is word-addressed, it does not use the two least significant bits. The output of the cache is a 4-byte word.

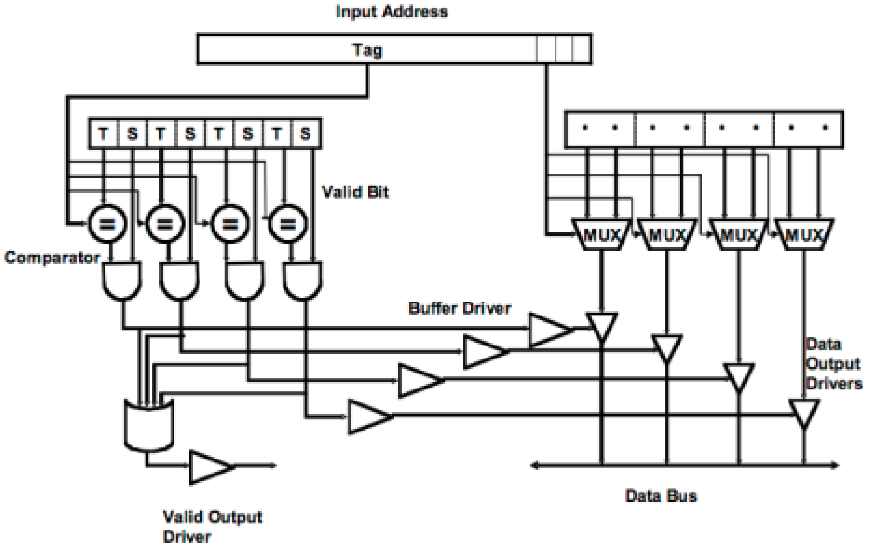


Figure 2.5-1: Victim cache datapath

Please complete Table 2.5-1 with delays across each element of the cache. Using the data you compute in Table 2.5-1, calculate the critical path delay through this cache (from when the Input Address is set to when both Valid Output Driver and the appropriate Data Output Driver are outputting valid data).

Component	Delay equation (ps)	FA(ps)
Comparator	$30\times$(# of tag bits) + 100	970
N-to-1 MUX	$50\times\log_2 N + 100$	150
Buffer driver	200	200
AND gate	100	100
OR gate	$50\times \log_2 N + 100$	200
Data output driver	$50\times$(associativity) + 100	300
Valid output driver	100	100

Table 2.5-1: Delay of each cache component

Critical Path Cache Delay:

Below, we evaluate the three major paths through the victim cache to find the critical path and cycle time. Note that the victim cache is fully-associative and uses 29-bit tags.

Candidate 1: Tag check
comparator → 2-in AND → 4-in OR → valid output driver
970 ps + 100 ps + 200 ps + 100 ps = 1370 ps

Candidate 2: Data select based on tag check
comparator → 2-in AND → buffer driver → data output driver
970 ps + 100 ps + 200 ps + 300 ps = 1570 ps

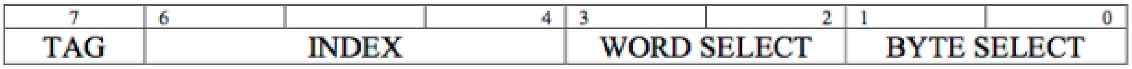
Candidate 3: Data readout
2-to-1 MUX → data output driver
200 ps + 300 ps = 500 ps

The critical path is the data select based on the tag match. The cycle time is 1570 ps.

Problem 4.B Victim Cache Behavior

Now we will study the impact of a victim cache on cache hit rate.
Our main L1 cache is a 128 byte, direct-mapped cache with 16 bytes per cache line. The cache is word (4-bytes) addressable.
The victim cache is similar to the one in Figure 2.5-1. It is a 32-byte fully associative cache with 16 bytes per cache line and is also word addressable. (Note that these parameters are different from 4.A.) It uses the first in first out (FIFO) replacement policy.

Please complete Table 2.5-2 showing a trace of memory accesses. In the table, each entry contains the tag of that line, or “inv”, if no data is present. You should only fill in elements in the table when a value changes. For simplicity, the addresses are only 8 bits. The first 3 lines of the table have been filled in for you. For your convenience, the address breakdown for access to the main cache is depicted below.



7	6		4	3	2	1	0
TAG	INDEX			WORD SELECT		BYTE SELECT	

Input Address	Main Cache (tag)									Victim Cache (tag)		
	L0	L1	L2	L3	L4	L5	L6	L7	Hit?	Way0	Way1	Hit?
	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv	-
0	0								N			N
80	1								N	0		N
4	0								N	8		Y
A0			1						N			N
10		0							N			N
C0					1				N			N
18		0							Y			
20			0						N		A	N
8C	1								N	0		Y
28			0						Y			
AC			1						N		2	Y
38				0					N			N
C4					1				Y			
3C				0					Y			
48					0				N	C		N
0C	0								N		8	N
24			0						N	A		N

Table 2.5-2: Memory access trace

Problem 4.C Average Memory Access Time

Assume **15%** of L1 misses are resolved in the victim cache. If retrieving data from the victim cache takes **4 cycles** and retrieving data from main memory takes **50 cycles**, by how many cycles does the victim cache improve the average memory access time? Assume that the L1 miss rate is **10%**.

AMAT = HitTime + L1MissRate * L1MissPenalty
AMAT2 = HitTime + L1MissRate * (VictimHitTime + (1 - VictimHitRate) * VictimMissPenalty)
VictimMissPenalty = L1MissPenalty = DRAMTime, since this is just time to get data from main memory
AMAT – AMAT2 = L1MissRate * (DRAMTime – VictimHitTime - (1 – VictimHitRate) * DRAMTime)
= 0.1 * (50 – 4 – 0.85 * 50)
= 0.1 * 3.5 = 0.35

Problem 5: Three C's of Cache Misses

Mark whether the following modifications will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. You can assume the baseline cache is set associative. **Explain your reasoning**.
For subparts where the outcome is ambiguous, pick one outcome and answer with reasonable assumptions and explanations.

	Compulsory Misses	Conflict Misses	Capacity Misses
Halving the line size (associativity and # sets constant) Halves capacity	Increase Shorter lines mean fewer adjacent elements are brought in with the first access to a given line.	Increase The program will access more cache lines in total, creating more opportunity for conflict misses.	Increase Capacity has been cut in half.
Doubling the number of sets (capacity and line size constant) Halves associativity	No effect Halving associativity doesn't change when lines are first brought into the cache	Increase Typically, lower associativity increases conflict misses, since there are fewer places to put the same element.	No effect Capacity does not change.
Adding good prefetching	Decrease Ideally, a good prefetcher can bring data in before we use it, avoiding compulsory misses.	Decrease With good prefetching, conflict misses should decrease, as the prefetcher will often bring lines that have been evicted back into the cache.	Decrease With good prefetching, capacity misses should decrease. In a situation where the working set simply won't fit, the prefetcher can dynamically bring lines in, "Just-In- Time," avoiding what would have been capacity misses.

Combine ICache and DCache into a single L1 cache with the combined capacity (associativity and line size constant)	No effect	May increase: New opportunities for conflicts between cache lines for data and cache lines for instructions are introduced	Decrease: Greater capacity

Problem 6: Memory Hierarchy Performance

Mark whether the following modifications will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. You can assume the baseline cache is set associative. **Explain your reasoning**.
For subparts where the outcome is ambiguous, pick one outcome and answer with reasonable assumptions and explanations.

	Hit Time	Miss Rate	Miss Penalty
Halving the line size (associativity and # sets constant) Halves capacity	Decreases The cache is now physically smaller, which overshadows the slightly increased tag check time (tag grows by 1 bit).	Increases Smaller capacity, less ability to take advantage of spatial locality within a single cache line (more compulsory misses).	Decreases Smaller lines can be brought in more quickly. OR No effect because cache already brings in critical word first.
Doubling the number of sets (capacity and line size constant) Halves associativity	Decreases # of sets increases, so tags get smaller. Fewer tags must be checked, and fewer ways have to be muxed outs.	Increases More conflict misses because associativity gets halved.	No effect This is dominated by the outer memory hierarchy
Adding good prefetching	No effect The prefetcher isn't on the hit path.	Decreases The whole purpose of a prefetcher is to reduce the miss rate by bringing in data ahead of time.	Good answer: no effect. May increase due to bandwidth pollution but we can(should) give a priority on cache misses over prefetch requests. May decrease because a prefetch can be inflight when a miss occurs (but this is unlikely).

Combine L1ICache and L1DCache into a single L1 cache with the combined capacity (associativity and line size constant)	Increase: If the cache is dual-ported, it will be slower than a single-ported cache If there is a single port, then frequently data accesses may stall for instruction accesses, or vice-versa	May Decrease Cache can more flexibly allocate space towards either data or instructions, depending on dynamic program behavior. May increase: Edge cases may cause more conflict misses between instruction and data accesses	No effect: This is dominated by outer memory hierarchy