

CS152 Computer Architecture and
Engineering

Caches and the Memory Hierarchy

Assigned
02/11/2026

Problem Set #2

Due February 20
@ 11:59:59PT

<https://inst.eecs.berkeley.edu/~cs152/sp26/>

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 [late submissions](#).

Name: _____

SID: _____

Collaborators (Name, SID):

Problem 1: Software Prefetching

Suppose we add software prefetching. Assume that each iteration of the inner loop takes 20 cycles (when there is a L1 miss). Of those 20 cycles, 15 cycles is the L1 miss penalty while 5 cycles is taken by the addition operation. Ignore any overheads from executing other instructions, including the software prefetch instruction. The prefetcher only prefetches when *prefetch_cond* becomes true (which is set by instructions that are not shown, and that you likewise do not need to model).

```
long sum = 0;
long matrix[64][64];
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        if (prefetch_cond) prefetch(&matrix[i][j]+OFFSET);
        sum += matrix[i][j];
    }
}
```

a) Assume your cache has 64-byte cache lines, what should OFFSET be set to in order to prefetch the correct value? The address calculation works on the granularity of bytes.

b) What should *prefetch_cond* be to minimize total number of cycles of execution?

c) Assume the matrix elements are stored in row-major order, how many cache misses will be saved by adding the software prefetching?

d) Why is software prefetching hard to make effective in practice? If we want to use hardware prefetcher instead, what kind of hardware prefetcher works well for this access pattern?

Problem 2: 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)			
Doubling the number of sets (capacity and line size constant)			
Combine ICache and DCache into a single L1 cache with the combined capacity (associativity and line size constant)			

Problem 3: Memory Hierarchy Performance

Problem 3.A

Victim Cache Behavior

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)			
Doubling the number of sets (capacity and line size constant)			
Combine L1ICache and L1DCache into a single L1 cache with the combined capacity (associativity and line size constant)			

Problem 3.B**World Class CPU**

Imagine you're tweaking your world class CPU from HW1, and you want to optimize your cache. Provide three unique (as in not from part A) optimizations you would make to your cache, and explain how/when it benefits cache performance. Feel free to be creative! Just provide a justification for your choice.

	Optimization	Impact
a)		
b)		
c)		

Problem 4 (OPTIONAL): 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.

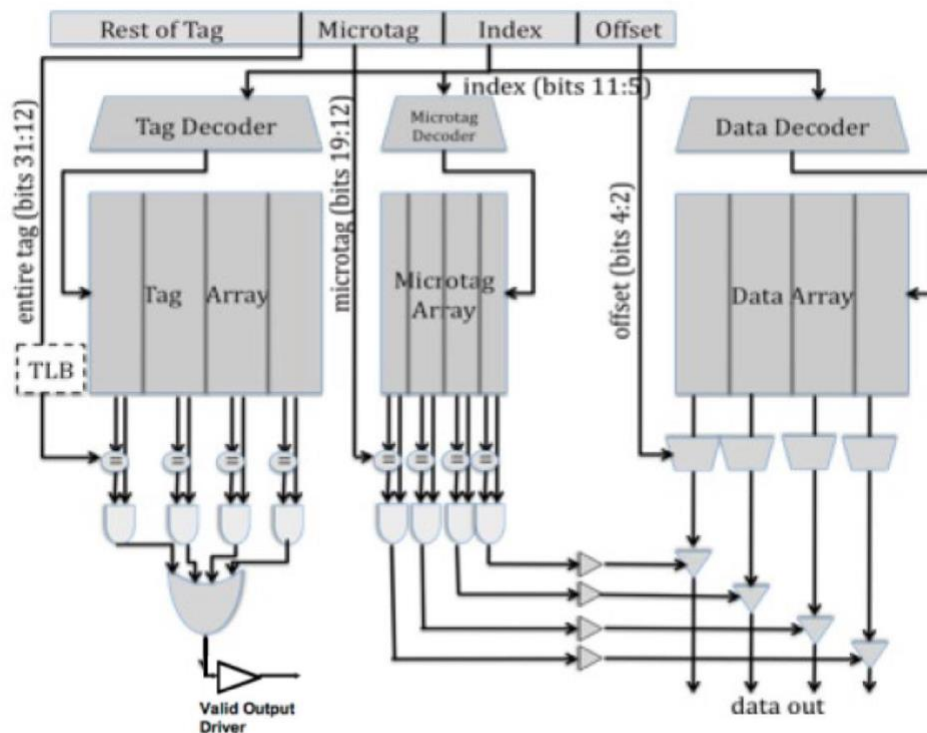


Figure 2.4-1: Microtagged cache datapath

**Problem 4.A
(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	$20 \cdot (\# \text{ of index bits}) + 100$	Tag	240	240
		Data	240	240
		Microtag		240
Memory array	$20 \cdot \log_2 (\# \text{ of rows}) + 20 \cdot \log_2 [(\# \text{ of bits in a row})] + 100$	Tag	380	380
		Data	440	440
		Microtag		340
Comparator	$20 \cdot (\# \text{ of tag bits}) + 100$	Tag	500	500
		Microtag		260
N-to-1 MUX	$50 \cdot \log_2 N + 100$		250	250
Buffer driver	200		200	200
Data output driver	$50 \cdot (\text{associativity}) + 100$		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)?

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

Problem 4.B (OPTIONAL)**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 hit times of 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)?

Problem 4.C (OPTIONAL)**Constraints**

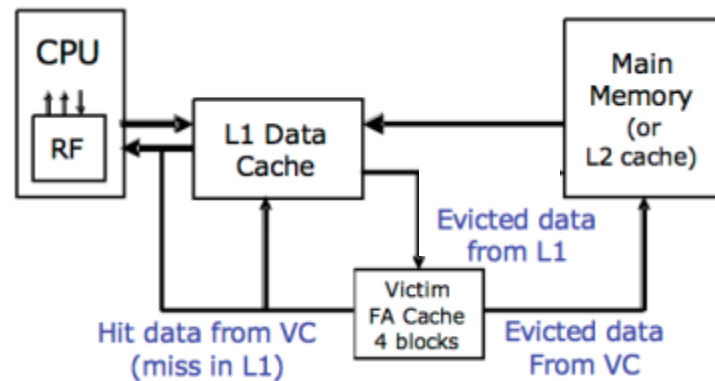
Microtags add an additional constraint to the cache: in a given cache set, all microtags must be unique. This constraint is necessary to avoid multiple microtag matches in the same set, which would prevent the cache from selecting the correct way.

- 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?

Problem 5 (OPTIONAL): 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 backup 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 5.A (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.

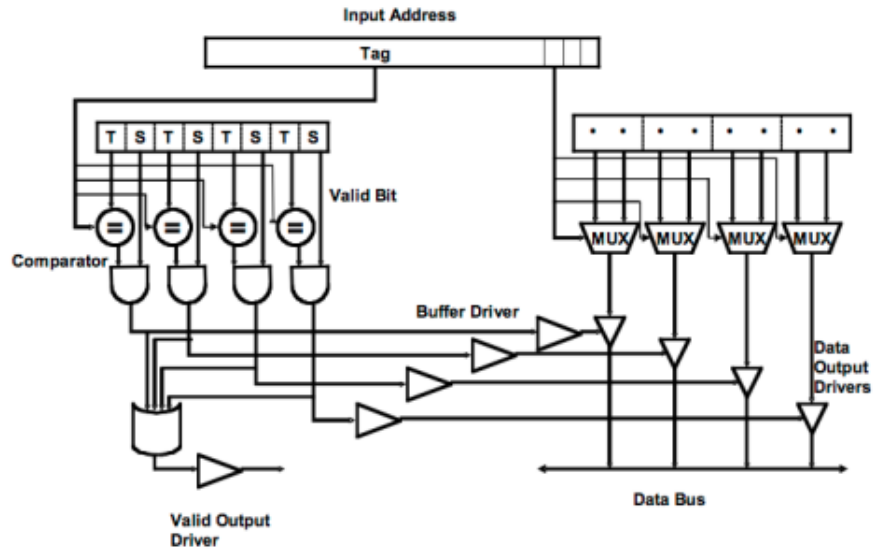


Figure 2.5-1: Victim cache datapath

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	$30 \cdot (\# \text{ of tag bits}) + 100$	
N-to-1 MUX	$50 \cdot \log_2 N + 100$	
Buffer driver	200	
AND gate	100	
OR gate	$50 \cdot \log_2 N + 100$	
Data output driver	$50 \cdot (\text{associativity}) + 100$	
Valid output driver	100	

Table 2.5-1: Delay of each cache component

Problem 5.B (OPTIONAL)

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												
10												
C0												
18												
20												
8C												
28												
AC												
38												
C4												
3C												
48												
0C												
24												

Table 2.5-2: Memory access trace

Problem 5.C (OPTIONAL)

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%**