Problem M3.1: Cache Access-Time & Performance

Here is the completed Table M3.1-1 for M3.1.A and M3.1.B.

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag	3400	3000
		Data	3400	3000
Memory array	200×$\log_2$(# of rows) + 200×$\log_2$(# of bits in a row) + 1000	Tag	4217	4250
		Data	5000	5000
Comparator	200×(# of tag bits) + 1000		4000	4400
N-to-1 MUX	500×$\log_2$ N + 1000		2500	2500
Buffer driver	2000			2000
Data output driver	500×(associativity) + 1000		1500	3000
Valid output driver	1000		1000	1000

Table M3.1-1: Delay of each Cache Component

Problem M3.1.A **Access time: DM**

To use the delay equations, we need to know how many bits are in the tag and how many are in the index. We are given that the cache is addressed by word, and that input addresses are 32-bit byte addresses; the two low bits of the address are not used.

Since there are 8 (2_3) words in the cache line, 3 bits are needed to select the correct word from the cache line.

In a 128 KB direct-mapped cache with 8 word (32 byte) cache lines, there are $4\times2_{10} = 2_{12}$ cache lines (128KB/32B). 12 bits are needed to address 2_{12} cache lines, so the number of index bits is 12. The remaining 15 bits (32 – 2 – 3 – 12) are the tag bits.

We also need the number of rows and the number of bits in a row in the tag and data memories. The number of rows is simply the number of cache lines (2_{12}), which is the same for both the tag and the data memory. The number of bits in a row for the tag memory is the sum of the number of tag bits (15) and the number of status bits (2), 17 bits total. The number of bits in a row for the data memory is the number of bits in a cache line, which is 256 (32 bytes × 8 bits/byte).

With 8 words in the cache line, we need an 8-to-1 MUX. Since there is only one data output driver, its associativity is 1.

Decoder (Tag) = 200 × (# of index bits) + 1000 = 200 × 12 + 1000 = 3400 ps
Decoder (Data) = 200 × (# of index bits) + 1000 = 200 × 12 + 1000 = 3400 ps

Memory array (Tag) = 200 × $\log_2$(# of rows) + 200 × $\log_2$(# bits in a row) + 1000

= 200 × $\log_2(2_{12})$ + 200 × $\log_2$(17) + 1000 ≈ 4217 ps

Memory array (Data) = 200 × $\log_2$(# of rows) + 200 × $\log_2$(# bits in a row) + 1000
= 200 × $\log_2(2_{12})$ + 200 × $\log_2$(256) + 1000 = 5000 ps

Comparator = 200 × (# of tag bits) + 1000 = 200 × 15 + 1000= 4000 ps

N-to-1 MUX = 500 × $\log_2$(N) + 1000 = 500 × $\log_2$(8) + 1000 = 2500 ps

Data output driver = 500 × (associativity) + 1000 = 500 × l + 1000 = 1500 ps

To determine the critical path for a cache read, we need to compute the time it takes to go through each path in hardware, and find the maximum.

Time to tag output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (valid output driver time)
≈ 3400 + 4217 + 4000 + 500 + 1000 = 13117 ps

Time to data output driver
= (data decode time) + (data memory access time) + (mux time) + (data output driver time)
= 3400 + 5000 + 2500 + 1500 = 12400 ps

The critical path is therefore the tag read going through the comparator. The access time is 13117 ps. At 150 MHz, it takes 0.013117 × 150, or 2 cycles, to do a cache access.

Problem M3.1.B **Access time: SA**

As in M3.1.A, the low two bits of the address are not used, and 3 bits are needed to select the appropriate word from a cache line. However, now we have a 128 KB 4-way set associative cache. Since each way is 32 KB and cache lines are 32 bytes, there are 2_{10} lines in a way (32KB/32B) that are addressed by 10 index bits. The number of tag bits is then (32 – 2 – 3 – 10), or 17.

The number of rows in the tag and data memory is 2_{10}, or the number of sets. The number of bits in a row for the tag memory is now quadruple the sum of the number of tag bits (17) and the number of status bits (2), 76 bits total. The number of bits in a row for the data memory is four times the number of bits in a cache line, which is 1024 (4 × 32 bytes × 8 bits/byte).

As in 1.A, we need an 8-to-1 MUX. However, since there are now four data output drivers, the associativity is 4.

Decoder (Tag) = 200 × (# of index bits) + 1000 = 200 × 10 + 1000 = 3000 ps
Decoder (Data) = 200 × (# of index bits) + 1000 = 200 × 10 + 1000 = 3000 ps

Memory array (Tag) = 200 × $\log_2$(# of rows) + 200 × $\log_2$(# bits in a row) + 1000

	$= 200 \times \log_2(2_{10}) + 200 \times \log_2(76) + 1000$	≈ 4250 ps
Memory array (Data)	$= 200 \times \log_2$(# of rows) $+ 200 \times \log_2$(# bits in a row) $+ 1000$	
	$= 200 \times \log_2(2_{10}) + 200 \times \log_2(1024) + 1000$	$= 5000$ ps
Comparator	$= 200 \times$ (# of tag bits) $+ 1000$	$= 200 \times 17 + 1000 = 4400$ ps
N-to-1 MUX	$= 500 \times \log_2(N) + 1000$	$= 500 \times \log_2(8) + 1000 = 2500$ ps
Data output driver	$= 500 \times$ (associativity) $+ 1000$	$= 500 \times 4 + 1000 = 3000$ ps

Time to valid output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (OR gate time) + (valid output driver time)
= 3000 + 4250 + 4400 + 500 + 1000 + 1000 = 14150 ps

There are two paths to the data output drivers, one from the tag side, and one from the data side. Either may determine the critical path to the data output drivers.

Time to get through data output driver through tag side
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (buffer driver time) + (data output driver)
= 3000 + 4250 + 4400 + 500 + 2000 + 3000 = 17150 ps

Time to get through data output driver through data side
= (data decode time) + (data memory access time) + (mux time) + (data output driver)
= 3000 + 5000 + 2500 + 3000 = 13500 ps

From the above calculations, it's clear that the critical path leading to the data output driver goes through the tag side.

The critical path for a read therefore goes through the tag side comparators, then through the buffer and data output drivers. The access time is 17150 ps. The main reason that the 4-way set associative cache is slower than the direct-mapped cache is that the data output drivers need the results of the tag comparison to determine which, if either, of the data output drivers should be putting a value on the bus. At 150 MHz, it takes 0.0175×150, or 3 cycles, to do a cache access.

It is important to note that the structure of cache we've presented here does not describe all the details necessary to operate the cache correctly. There are additional bits necessary in the cache which keeps track of the order in which lines in a set have been accessed. We've omitted this detail for sake of clarity.

Problem M3.1.C **Miss-rate analysis**

D-map									
	line in cache								hit?
Address	L0	L1	L2	L3	L4	L5	L6	L7	
110	inv	11	inv	inv	inv	inv	inv	inv	no
136				13					no
202	20								no
1A3			1A						no
102	10								no
361							36		no
204	20								no
114									yes
1A4									yes
177								17	no
301	30								no
206	20								no
135									yes

	D-map
Total Misses	10
Total Accesses	13

4-way									LRU
	line in cache								Hit?
Address	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	inv	inv	inv	11	inv	inv	inv	No
136						13			No
202	20								No
1A3		1A							No
102			10						No
361				36					No
204									Yes
114									Yes
1A4									Yes
177							17		No
301			30						No
206									Yes
135									Yes

	4-way LRU
Total Misses	8
Total Accesses	13

~~4-way~~ **Address**									FIFO Hit?
	line in cache								
	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	Inv	inv	inv	11	inv	inv	Inv	No
136						13			No
202	20								No
1A3		1A							No
102			10						No
361				36					No
204									Yes
114									Yes
1A4									Yes
177							17		No
301	30								No
206		20							No
135									Yes

	4-way FIFO
Total Misses	9
Total Accesses	13

Problem M3.1.D **Average latency**

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.

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

For the direct-mapped cache, the average memory access latency would be (2 cycles) + (10/13) × (20 cycles) = 17.38 ≈ 18 cycles.

For the LRU set associative cache, the average memory access latency would be (3 cycles) + (8/13) × (20 cycles) = 15.31 ≈ 16 cycles.

The set associative cache is better 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 the {20} block instead of the {10} block during the 12th access, because the {20}

block has been in the cache longer even though the {10} was least recently used, whereas the LRU policy took advantage of temporal/spatial locality.

LRU doesn't always have lower miss rate than FIFO. Consider the following counter example: A sequence accesses 3 separate memory locations A,B and C in the order of A, B, A, C, B, B, B, …. When this sequence is executed on a processor employing a fully-associative cache with 2 cache lines and LRU replacement policy, the execution ends up with 4 misses. On the other hand, the same sequence will only produces 3 misses if the cache uses FIFO replacement policy. (We assume the cache is empty at the beginning of the execution).

Problem M3.2: Victim Cache Evaluation

Problem M3.2.A **Baseline Cache Design**

Component	Delay equation (ps)		FA (ps)
Comparator	200×(# of tag bits) + 1000		6800
N-to-1 MUX	500×$\log_2$ N + 1000		1500
Buffer driver	2000		2000
AND gate	1000		1000
OR gate	500		500
Data output driver	500×(associativity) + 1000		3000
Valid output driver	1000		1000

Table M3.2-1

The **Input Address** has 32 bits. The bottom two bits are discarded (cache is word-addressable) and bit 2 is used to select a word in the cache line. Thus the **Tag** has 29 bits. The **Tag+Status** line in the cache is 31 bits.

The **MUX**es are 2-to-1, thus N is 2. The associativity of the **Data Output Driver** is 4 – there are four drivers driving each line on the common **Data Bus**.

Delay to the **Valid Bit** is equal to the delay through the **Comparator**, **AND** gate, **OR** gate, and **Valid Output Driver**. Thus it is 6800 + 1000 + 500 + 1000 = 9300 ps.

Delay to the **Data Bus** is delay through MAX ((**Comparator**, **AND** gate, **Buffer Driver**), (**MUX**)), **Data Output Drivers**. Thus it is MAX (6800 + 1000 + 2000, 1500) + 3000 = MAX (9800, 1500) + 3000 = 9800 + 3000 = 12800 ps.

Critical Path Cache Delay: 12800 ps

Problem M3.2.B **Victim Cache Behavior**

Input Address	Main Cache									Victim Cache		
	L0	L1	L2	L3	L4	L5	L6	L7	Hit?	Way0	Way1	Hit?
	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv	-
00	0								N			N
80	8								N	0		N
04	0								N	8		Y
A0			A						N			N
10		1							N			N
C0					C				N			N
18									Y			N
20			2						N		A	N
8C	8								N	0		Y
28									Y			N
AC			A						N		2	Y
38				3					N			N
C4									Y			N
3C									Y			N
48					4				N	C		N
0C	0								N		8	N
24			2						N	A		N

Table M3.2-2

Problem M3.2.C **Average Memory Access Time**

15% of accesses will take 50 cycles less to complete, so the average memory access improvement is 0.15 * 50 = 7.5 cycles.

Problem M3.3: Loop Ordering

Problem M3.3.A

Each element of the matrix can only be mapped to a particular cache location because the cache here is a Direct-mapped data cache. *Matrix A* has 64 columns and 128 rows. Since each row of matrix has 64 32-bit integers and each cache line can hold 8 words, each row of the matrix fits exactly into eight (64÷8) cache lines as the following:

0	A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]	A[0][5]	A[0][6]	A[0][7]
1	A[0][8]	A[0][9]	A[0][10]	A[0][11]	A[0][12]	A[0][13]	A[0][14]	A[0][15]
2	A[0][16]	A[0][17]	A[0][18]	A[0][19]	A[0][20]	A[0][21]	A[0][22]	A[0][23]
3	A[0][24]	A[0][25]	A[0][26]	A[0][27]	A[0][28]	A[0][29]	A[0][30]	A[0][31]
4	A[0][32]	A[0][33]	A[0][34]	A[0][35]	A[0][36]	A[0][37]	A[0][38]	A[0][39]
5	A[0][40]	A[0][41]	A[0][42]	A[0][43]	A[0][44]	A[0][45]	A[0][46]	A[0][47]
6	A[0][48]	A[0][49]	A[0][50]	A[0][51]	A[0][52]	A[0][53]	A[0][54]	A[0][55]
7	A[0][56]	A[0][57]	A[0][58]	A[0][59]	A[0][60]	A[0][61]	A[0][62]	A[0][63]
8	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	A[1][6]	A[1][7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Loop A accesses memory sequentially (each iteration of *Loop A* sums a row in *matrix* `A`), an access to a word that maps to the first word in a cache line will miss but the next seven accesses will hit. Therefore, *Loop A* will only have compulsory misses (128×64÷8 or 1024 misses).

The consecutive accesses in *Loop B* will use every eighth cache line (each iteration of *Loop B* sums a column in *matrix* `A`). Fitting one column of matrix A, we would need 128×8 or 1024 cache lines. However, our 4KB data cache with 32B cache line only has 128 cache lines. When Loop B accesses a column, all the data that the previous iteration might have brought in would have already been evicted. Thus, every access will cause a cache miss (64×128 or 8192 misses).

The number of cache misses for Loop A: 1024

The number of cache misses for Loop B: 8192

Problem M3.3.B

Since *Loop A* accesses memory sequentially, we can overwrite the cache lines that were previous brought in. *Loop A* will only require 1 cache line to run without any cache misses other than compulsory misses.

For *Loop B* to run without any cache misses other than compulsory misses, the data cache needs to have the capacity to hold one column of matrix A. Since the consecutive accesses in *Loop B* will use every eighth cache line and we have 128 elements in a *matrix A* column, *Loop B* requires 128×8 or 1024 cache lines.

Data-cache size required for Loop A: 1 cache line(s)

Data-cache size required for Loop B: 1024 cache line(s)

Problem M3.3.C

Loop A still only has compulsory misses (128×64÷8 or 1024 misses).

Because of the fully-associative data cache, *Loop B* now can fully utilize the cache and the consecutive accesses in *Loop B* will no longer use every eighth cache line. Fitting one column of *matrix A,* we now would only need 128 cache lines. Since 4KB data cache with 8-word cache lines has 128 cache lines, *Loop B* only has compulsory misses (128×(64÷8) or 1024 misses).

The number of cache misses for Loop A: 1024

The number of cache misses for Loop B: 1024

Problem M3.4: Cache Parameters

Problem M3.4.A

TRUE. Since cache size is unchanged, the line size doubles, the number of tag entries is halved.

Problem M3.4.B

FALSE. The total number of lines across all sets is still the same, therefore the number of tags in the cache remain the same.

Problem M3.4.C

TRUE. Doubling the capacity increases the number of lines from N to 2N. Address i and address i+N now map to different entries in the cache and hence, conflicts are reduced.

Problem M3.4.D

FALSE. The number of lines doubles but the line size remains the same. So the compulsory "cold-start" misses stays the same.

Problem M3.4.E

TRUE. Doubling the line size causes more data to be pulled into the cache on a miss. This exploits spatial locality as subsequent loads to different words in the same cache line will hit in the cache reducing compulsory misses.

Problem M3.5: Microtags

Problem M3.5.A

A direct-mapped cache can forward data to the CPU before checking the tags for a hit or a miss. A set-associative cache has to first compare cache tags to select the correct way from which to forward data to the CPU.

Problem M3.5.B

tag	Index	offset

# of bits in the tag: ____21_____

# of bits in the index: _____6_____

# of bits in the offset: _____5_____

32-byte line requires 5 bits to select the correct byte.
An 8KB, 4-way cache has 2KB in each way, and each way holds 2KB/32B=64 lines, so we need 6 index bits.
The remaining 32-6-5=21 bits are the tag.

Problem M3.5.C

If the loTags are not unique, then multiple ways can attempt to drive data on the tristate bus out to the CPU causing bus contention.

(It is possible to have a scheme that speculatively picks one of the ways when there is as match in loTags, but this would require additional cross-way logic that would slow the design down, and would also incur extra misses when the speculation was wrong.)

Problem M3.5.D

The loTag has to be unique across ways, and so in a 4-way cache with 2-bit tags the tags would never be able to hold addresses that were different from a direct-mapped cache of the same capacity. The conflict misses would therefore be identical.

Problem M3.5.E

When a new line is brought into the cache, any existing line in the set with the same loTag must be chosen as the victim. If there is no line with the same loTag, any conventional replacement policy can be used.

Problem M3.5.F

No. The full tag check is required to determine whether the write is a hit to the cached line.

Problem M3.5.G

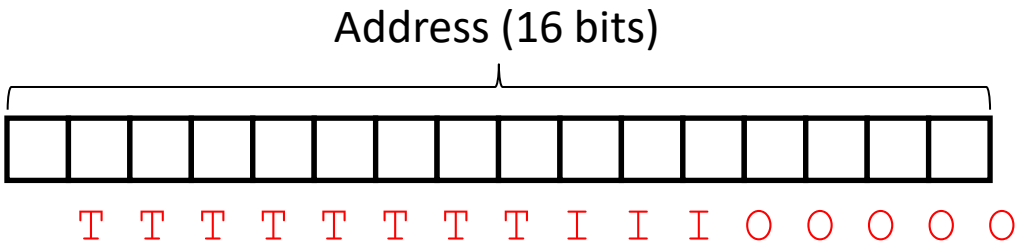
A 16KB page implies 14 untranslated address bits. An 8KB, 4-way cache requires 11 index+offset bits, leaving 3 untranslated bits for loTag.

Problem M3.5.H

If the loTags include translated virtual address bits, then each cache line must store the physical page number (PPN) as the hiTag. An access will hit if loTag matches, and the PPN in hiTag matches. The replacement policy has to maintain two invariants: 1) no two lines in a set have the same loTag bits and 2) no two lines have the same PPN. If two lines had the same PPN, there might be a virtual address alias. Because a new line might have the same loTag as an existing line, and also the same PPN as a different line, two lines might have to be evicted to bring in one new line.
A slight improvement is to only evict a line with the same PPN if the untranslated part of loTag is identical. If the untranslated bits are different, the two lines cannot be aliases.

Problem M3.6: Caches (Spring 2014 Quiz 1, Part C)

Problem M3.6.A



Divide the bits of the address according to how they are used to access the cache (tag, index, offset). Drawn above (letters). Block size is 32 bytes, so there are five offset bits. We have 8 lines in a direct mapped organization (as indicated by diagram), so we need three index bits. The remaining 8 bits constitute the tag.

What exactly is contained in the cache tags? (Include all bits necessary for correct operation of the cache as discussed in lecture.) The tag bits of address and valid and dirty bits (dirty not required since lecture didn't cover cache writes). Replacement policy bits are not present because the cache is direct mapped.

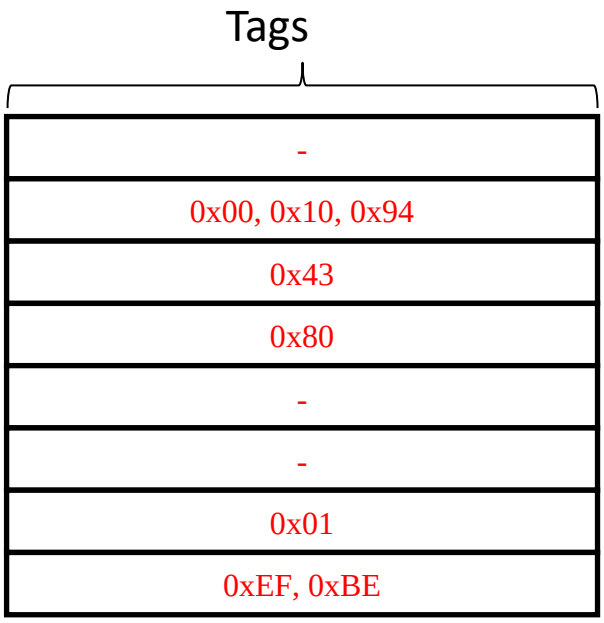
How many bits in total are needed to implement the level 1 data cache? The cache consists of tag and data arrays, or 8 lines x (256 bytes/block + 10 bits/tag) = 2128 bits.

Problem M3.6.B

Suppose the processor accesses the following data addresses starting with an empty cache:

```
0x0028: 0000 0000 0010 1000 Miss
0x102A: 0001 0000 0010 1010 Miss
0x9435: 1001 0100 0011 0101 Miss
0xEFF4: 1110 1111 1111 0100 Miss
0xBEEF: 1011 1110 1110 1111 Miss
0x4359: 0100 0011 0101 1001 Miss
0x01DE: 0000 0001 1101 1110 Miss
0x8075: 1000 0000 0111 0101 Miss
0x9427: 1001 0100 0010 0111  Hit
```

What would the level 1 data cache tags look like after this sequence? How many hits would there be in the level 1 data cache? *(Don't worry about filling in the Data column – we didn't give you the data!)*



We did not knock off points for not showing status bits, although an exact solution would show which lines were dirty and valid. (Dirty is ambiguous since the problem doesn't specify whether accesses are reads or writes.)

Problem M3.6.C

Suppose that the level 1 data cache has a hit rate of 40% on your application, an access time of a single cycle, and a miss penalty to memory of forty cycles. What is the average memory access time?

AMAT = hit time + miss rate * miss penalty
= 1 + (1 - 0.4) * 40
= 25 cycles

Or, equivalently:

AMAT = hit rate * hit time + miss rate * miss time
= 0.4 * 1 + 0.6 * (1 + 40)
= 25 cycles

You aren't happy with your memory performance, so you decide to add a level two cache. Suppose the level two cache has a hit rate of 50%. What access time must the level two cache have for this to be a good design (ie, reduce AMAT)?

The L2 lies between the L1 and memory, and is only accessed if the L1 misses. To get to memory, you therefore need to miss in the L1 *then* miss in the L2 *then* go to memory (all sequentially).

There are two ways to solve this problem. The first is to realize that if the L2 improves the system's average memory access time, then it must improve the AMAT of accesses into it (ignoring whatever happens at the L1). In other words, each level of the cache hierarchy can be modeled independently of levels below it. This simplifies the problem to solving for the L2 access time such that:

L2 AMAT < Memory time
L2 access time + L2 miss rate * Memory time < Memory time
L2 access time + 0.5 * 40 < 40
L2 access time < 20

If instead you model the full cache hierarchy, the L2 only sees lines that the miss in the L1. Thus with an L2, the L1's miss penalty is the average memory access time of the L2. So the equation is:

L1 access time + L1 miss rate * L2 AMAT < L1 access time + L1 miss rate * Memory time
L2 AMAT < (L1 miss rate * Memory time + L1 access time – L1 access time) / L1 miss rate
L2 AMAT < Memory time

Now we are back to the formula we derived first by solving the L2 independently.