CS 351 Homework 12 Solutions

Problem 1: Cache Tracing - Direct Mapped - 2B blocks (25 points)

You have a direct-mapped cache with 16 B of data (not counting metadata). Each cache block is 2 B (again, not counting metadata). Main memory addresses are 8 bits.
Assume that the cache is initially empty (all blocks are invalid), and fill in the table below to trace the following sequence of read accesses.

Part A: Tracing individual read accesses (16 points, 2 per row)

Complete the following table showing how each address is split into tag, index, and offset, as well as whether it is a hit or a miss and which addresses' data, if any, it evicts. These accesses are starting in sequential order, with the first one accessing an empty cache.

Part B: Final cache state (9 points, EMRN)

Draw the final state of the cache, clearly labeling each block and set. Your drawing should clearly show all relevant valid and tag bits in the cache.

Problem 1 Solutions (all parts)

The number of offset bits depends only on the block size. Since there are 2 bytes per block, the number of offset bits is log_2(2) = 1.
Since this is a 16B cache with 2-byte blocks, it has 16/2 = 8 blocks.
In a direct-mapped cache, you use index bits to select a unique block for each address. So we need log_2(8)=3 index bits, and the remaining bits of the address will be the tag. This means we have enough information to split up the address and draw the initial state of the cache.

Problem 1 Address Breakdown

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Address(es) evicted (if any)**
1010 1111	*1010*	*111*	*1*		
1011 1111	*1011*	*111*	*1*		
1010 1111	*1010*	*111*	*1*		
1011 1110	*1011*	*111*	*0*		
1010 1111	*1010*	*111*	*1*		
1011 1111	*1011*	*111*	*1*		
1010 1110	*1010*	*111*	*0*		
1011 1111	*1011*	*111*	*1*		

Since these accesses all map to the same cache block (block 111), we can look forward to some thrashing and conflict misses.

Problem 1 Initial Cache State

Here are the 8 blocks with their data and metadata. Note that the index bits are not actually stored in the cache.

Index (not stored)	Data[0]	Data[1]	Valid	Tag
000			0	
001			0	
010			0	
011			0	
100			0	
101			0	
110			0	
111			0	

Problem 1: First Access (1010 1111)

The first access is to block 111, which is invalid, so this is a miss. We bring this byte and its neighbor at address 1010 1110 into block 111.

Index (not stored)	Data[0]	Data[1]	Valid	Tag
000			0	
001			0	
010			0	
011			0	
100			0	
101			0	
110			0	
111	Data from address 1010 1110	Data from address 1010 1111	1	1010

Problem 1: Second Access (1011 1111)

To see if this is a hit, we check block 111. Alas, the tag of 1010 doesn't match our tag of 1011. So we evict the data that is currently in block 111 and bring in this byte and its neighbor at address 1011 1110:

Index (not stored)	Data[0]	Data[1]	Valid	Tag
000			0	
001			0	
010			0	
011			0	
100			0	
101			0	
110			0	
111	Data from address 1011 1110	Data from address 1011 1111	1	1011

Problem 1: Remaining Accesses

*…and those first 2 accesses are representative of how this is going to go. Our third access will be a miss because the tag of 1010 doesn't match, and then we'll bring the data from the first access back in, kicking out the second. Our fourth access will look just like the second access, and so on – we are **thrashing** between these two different addresses, resulting in a 0% overall hit rate even though we have plenty of temporal locality and the cache has plenty of open spots.*

Problem 1A Final Answer

Address	Tag	Index	Offset	H/M?	Address(es) evicted (if any)
1010 1111	*1010*	*111*	*1*	*M*	–
1011 1111	*1011*	*111*	*1*	*M*	*1010 1110 and 1010 1111*
1010 1111	*1010*	*111*	*1*	*M*	*1011 1110 and 1011 1111*
1011 1110	*1011*	*111*	*0*	*M*	*1010 1110 and 1010 1111*
1010 1111	*1010*	*111*	*1*	*M*	*1011 1110 and 1011 1111*
1011 1111	*1011*	*111*	*1*	*M*	*1010 1110 and 1010 1111*
1010 1110	*1010*	*111*	*0*	*M*	*1011 1110 and 1011 1111*
1011 1111	*1011*	*111*	*1*	*M*	*1010 1110 and 1010 1111*

Problem 1B: Final answer

The final access is to address 1011 1111, so this address and its neighbor will be the final value in the highly contested block #111.

Index (not stored)	Data[0]	Data[1]	Valid	Tag
000			0	
001			0	
010			0	
011			0	
100			0	
101			0	
110			0	
111	Data from address 1011 1110	Data from address 1011 1111	1	1011

Problem 2: Cache Tracing - Direct Mapped - 4B blocks (25 points)

You have a direct-mapped cache with 16 B of data (not counting metadata). Each cache block is 4 B (again, not counting metadata). Main memory addresses are 8 bits.
Assume that the cache is initially empty (all blocks are invalid), and fill in the table below to trace the following sequence of read accesses.

Part A: Tracing individual read accesses (16 points, 2 per row)

Complete the following table showing how each address is split into tag, index, and offset, as well as whether it is a hit or a miss and which addresses' data, if any, it evicts. These accesses are starting in sequential order, with the first one accessing an empty cache.

Address	Tag	Index	Offset	H/M?	Address(es) evicted (if any)
1010 1111					
1011 1111					
1010 1111					
1011 1110					
1010 1111					
1011 1111					
1010 1110					
1011 1111					

Part B: Final cache state (9 points, EMRN)

Draw the final state of the cache, clearly labeling each block and set. Your drawing should clearly show all relevant valid and tag bits in the cache.

Problem 2 Solutions (all parts)

The number of offset bits depends only on the block size. Since there are 4 bytes per block, the number of offset bits is log_2(2) = 2.
Since this is a 16B cache with 4-byte blocks, it has 16/4 = 4 blocks.
In a direct-mapped cache, you use index bits to select a unique block for each address. So we need log_2(4)=2 index bits, and the remaining bits of the address will be the tag. This means we have enough information to split up the address and draw the initial state of the cache.

Problem 2 Address Breakdown

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Address(es) evicted (if any)**
1010 1111	*1010*	*11*	*11*		
1011 1111	*1011*	*11*	*11*		
1010 1111	*1010*	*11*	*11*		
1011 1110	*1011*	*11*	*10*		
1010 1111	*1010*	*11*	*11*		
1011 1111	*1011*	*11*	*11*		
1010 1110	*1010*	*11*	*10*		
1011 1111	*1011*	*11*	*11*		

And once again, everybody is fighting for the same block – this time, it's block 11. So we can expect this go to about as badly as Problem 1.

Problem 2 Initial Cache State

Here are the 4 blocks with their data and metadata. Note that the index bits are not actually stored in the cache.

Index (not stored)	**Data[00]**	**Data[01]**	**Data[10]**	**Data[11]**	**Valid**	**Tag**
00					0	
01					0	
10					0	
11					0	

Problem 2: First Access (1010 1111)

The first access is to block 11, which is invalid, so this is a miss. We bring this byte and its neighbors at addresses 101011{00,01,10,11} into block 11.

Index (not stored)	Data[00]	Data[01]	Data[10]	Data[11]	Valid	Tag
00					0	
01					0	
10					0	
11	Data from 1010 1100	Data from 1010 1101	Data from 1010 1110	Data from 1010 1111	1	1010

Problem 2: Second Access (1011 1111) and Final Cache State

To see if this is a hit or a miss, we check the valid and tag bits of block 11. Alas, the tags don't match, so the current data gets evicted and the new data gets brought in.
From here, the two groups of 4 bytes will keep kicking each other out of this block. The last access is to address 1011 1111, so this will be the last block standing, and so this will also be the final cache state.

Index (not stored)	Data[00]	Data[01]	Data[10]	Data[11]	Valid	Tag
00					0	
01					0	
10					0	
11	Data from 1011 1100	Data from 1011 1101	Data from 1011 1110	Data from 1011 1111	1	1011

Problem 2A Final Answer

Address	Tag	Index	Offset	H/M?	Address(es) evicted (if any)
1010 1111	*1010*	*11*	*11*	*M*	–
1011 1111	*1011*	*11*	*11*	*M*	*1010 1100, 1010 1101, 1010 1110, 1010 1111*
1010 1111	*1010*	*11*	*11*	*M*	*1011 1100, 1011 1101, 1011 1110, 1011 1111*
1011 1110	*1011*	*11*	*10*	*M*	*1010 1100, 1010 1101, 1010 1110, 1010 1111*
1010 1111	*1010*	*11*	*11*	*M*	*1011 1100, 1011 1101, 1011 1110, 1011 1111*
1011 1111	*1011*	*11*	*11*	*M*	*1010 1100, 1010 1101, 1010 1110, 1010 1111*
1010 1110	*1010*	*11*	*10*	*M*	*1011 1100, 1011 1101, 1011 1110, 1011 1111*

1011 1111	*1011*	*11*	*11*	*M*	*1010 1100, 1010 1101, 1010 1110, 1010 1111*

Problem 3: Cache Tracing - 2-Way SA with 1B blocks (25 points)

You have a two-way set-associative cache with 16 B of data (not counting metadata). **Each cache block is 1 B (again, not counting metadata).** Main memory addresses are 8 bits, and precise LRU is used as the replacement policy between the two blocks in a set.
Assume that the cache is initially empty (all blocks are invalid), and fill in the table below to trace the following sequence of read accesses.

Part A: Tracing individual read accesses (16 points, 2 per row)

Complete the following table showing how each address is split into tag, index, and offset, as well as whether it is a hit or a miss and which addresses' data, if any, it evicts. These accesses are starting in sequential order, with the first one accessing an empty cache.

Part B: Final cache state (9 points, EMRN)

Draw the final state of the cache, clearly labeling each block and set. Your drawing should clearly show all relevant valid, tag, and LRU bits in the cache.

Problem 3 Solutions (all parts)

The number of offset bits depends only on the block size. Since there is 1 byte per block, the number of offset bits is log_2(1) = 0.
Since this is a 16B cache with 1-byte blocks, it has 16/1 = 16 blocks. In a 2-way set-associative cache, these are split into 8 sets of 2 blocks per set.
In a set-associative cache, you use index bits to select a unique SET for each address. So we need log_2(8)=3 index bits, and the remaining bits of the address will be the tag. This means we have enough information to split up the address and draw the initial state of the cache.

Problem 3 Address Breakdown

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Address(es) evicted (if any)**
1010 1111	*10101*	*111*	–		
1011 1111	*10111*	*111*	–		
1010 1111	*10101*	*111*	–		
1011 1110	*10111*	*110*	–		
1010 1111	*10101*	*111*	–		
1011 1111	*10111*	*111*	–		
1010 1110	*10101*	*110*	–		
1011 1111	*10111*	*111*	–		

Here, we finally have 2 different index values involved: 110 and 111. Furthermore, there are 2 blocks in each of these sets, so MAYBE we won't see as may conflict misses as we did with the direct-mapped cache.

Problem 3 Initial Cache State

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110		0		←		0	
111		0		←		0	

Problem 3 First Access (1010 1111)

The first access goes to set 111. Since neither block is valid, that's a miss. Since the LRU bits point to Way A, we populate the left block in the set and flip the LRU to point to the right block:

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110		0		←		0	
111	Data from 1010 1111	1	10101	→		0	

Problem 3 Second Access (1011 1111)

The first access goes to set 111. Neither of the blocks in that set is valid with a tag matching 10111, so this is a miss. Following the LRU bit, we bring the new data into Way B and flip the LRU back to Way A.

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110		0		←		0	
111	Data from 1010 1111	1	10101	←	Data from 1011 1111	1	10111

Problem 3 Third Access (1010 1111)

This access also sends us to set 111. The left block (Way A) in that set is valid, and the tag matches! IT'S AN END OF SEMESTER MIRACLE! This is a hit. We update LRU to point away from this block.

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110		0		←		0	
111	Data from 1010 1111	1	10101	←	Data from 1011 1111	1	10111

Problem 3 Fourth Access (1011 1110)

This access sends us to set 110. Neither block is valid, so that's a miss. We bring the data into Way A and flip the LRU bits to Way B.

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110	Data from 1011 1110	1	10111	→		0	
111	Data from 1010 1111	1	10101	←	Data from 1011 1111	1	10111

Problem 3 Fifth Access (1010 1111)

This access sends us to set 111. This is a hit, and we just point the LRU arrow back to Way B.

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110	Data from 1011 1110	1	10111	→		0	
111	Data from 1010 1111	1	10101	→	Data from 1011 1111	1	10111

Problem 3 Sixth Access (1011 1111)

This access sends us to set 111. This is a hit, and we flip the LRU arrow back to Way A.

Index (not stored)	Way A Data	Way A Valid	Way A Tag	LRU	Way B Data	Way B Valid	Way B Tag

000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110	Data from 1011 1110	1	10111	→		0	
111	Data from 1010 1111	1	10101	←	Data from 1011 1111	1	10111

Problem 3 Seventh and Eighth Accesses (1010 1110 and 1011 1111)

This access sends us to set 110. This is a miss, since this doesn't match the tag of the existing valid block, so we bring this data into Way B and flip the LRU bit.
The final access is a hit to set 111, Way B. The LRU bit is already pointed away from Way B, so the final access does not change the cache.

Index (not stored)	**Way A Data**	**Way A Valid**	**Way A Tag**	**LRU**	**Way B Data**	**Way B Valid**	**Way B Tag**
000		0		←		0	
001		0		←		0	
010		0		←		0	
011		0		←		0	
100		0		←		0	
101		0		←		0	
110	Data from 1011 1110	1	10111	←	Data from 1010 1110	1	10101
111	Data from 1010 1111	1	10101	←	Data from 1011 1111	1	10111

Problem 3A Final Answer

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Address(es) evicted (if any)**
1010 1111	*10101*	*111*	–	*M*	

1011 1111	*10111*	*111*	–	*M*	
1010 1111	*10101*	*111*	–	*H*	
1011 1110	*10111*	*110*	–	*M*	
1010 1111	*10101*	*111*	–	*H*	
1011 1111	*10111*	*111*	–	*H*	
1010 1110	*10101*	*110*	–	*M*	
1011 1111	*10111*	*111*	–		

Problem 4: Cache Tracing - 2-Way SA with 2B blocks (25 points)

You have a two-way set-associative cache with 16 B of data (not counting metadata). **Each cache block is now 2 B (again, not counting metadata).** Main memory addresses are 8 bits, and precise LRU is used as the replacement policy between the two blocks in a set.

Assume that the cache is initially empty (all blocks are invalid), and fill in the table below to trace the following sequence of read accesses.

Part A: Tracing individual read accesses (16 points, 2 per row)

Complete the following table showing how each address is split into tag, index, and offset, as well as whether it is a hit or a miss and which addresses' data, if any, it evicts. These accesses are starting in sequential order, with the first one accessing an empty cache.

Part B: Final cache state (9 points, EMRN)

Draw the final state of the cache, clearly labeling each block and set. Your drawing should clearly show all relevant valid, tag, and LRU bits in the cache.

Problem 4 Solutions (all parts)

The number of offset bits depends only on the block size. Since there is 1 byte per block, the number of offset bits is log_2(2) = 1.
Since this is a 16B cache with 2-byte blocks, it has 16/2 = 8 blocks. In a 2-way set-associative cache, these are split into 4 sets of 2 blocks per set.
In a set-associative cache, you use index bits to select a unique **set** *for each address. So we need log_2(4)=2 index bits, and the remaining bits of the address will be the tag. This means we have enough information to split up the address and draw the initial state of the cache.*

Problem 4 Address Breakdown

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Address(es) evicted (if any)**
1010 1111	*10101*	*11*	*1*		

1011 1111	*10111*	*11*	*1*		
1010 1111	*10101*	*11*	*1*		
1011 1110	*10111*	*11*	*0*		
1010 1111	*10101*	*11*	*1*		
1011 1111	*10111*	*11*	*1*		
1010 1110	*10101*	*11*	*0*		
1011 1111	*10111*	*11*	*1*		

Now, everybody is fighting over the two blocks in set 11. Let's see how that goes.

Problem 4 Initial Cache State

Index (not stored)	**Way A Data[0]**	**Way A Data[1]**	**Way A Valid**	**Way A Tag**	**LRU**	**Way B Data[0]**	**Way B Data[1]**	**Way B Valid**	**Way B Tag**
00			0		←			0	
01			0		←			0	
10			0		←			0	
11			0		←			0	

Problem 4 First Access (1010 1111)

The first access goes to set 11. Neither block is valid, so this is a miss. The data will populate Way A of this block, and the LRU bits will flip to B.

Index (not stored)	**Way A Data[0]**	**Way A Data[1]**	**Way A Valid**	**Way A Tag**	**LRU**	**Way B Data[0]**	**Way B Data[1]**	**Way B Valid**	**Way B Tag**
00			0		←			0	
01			0		←			0	
10			0		←			0	
11	1010 1110	1010 1111	1	10101	→			0	

Problem 4 Second through Eighth Accesses

The second access goes to set 11. There is no valid block whose tags match, so it will populate Way B and flip the LRU bits.
The third access is a hit to Way A, so the LRU bit will flip to Way B. But then the fourth access is a hit to Way B, and it will flip right back.
All the subsequent accesses will be hits and will keep flipping the bit back and forth.

Index (not stored)	**Way A Data[0]**	**Way A Data[1]**	**Way A Valid**	**Way A Tag**	**LRU**	**Way B Data[0]**	**Way B Data[1]**	**Way B Valid**	**Way B Tag**
00			0		←			0	
01			0		←			0	
10			0		←			0	
11	1010 1110	1010 1111	1	10101	←	1011 1110	1011 1111	0	10111

Problem 4 Final Answer

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Address(es) evicted (if any)**
1010 1111	*10101*	*11*	*1*	*M*	
1011 1111	*10111*	*11*	*1*	*M*	
1010 1111	*10101*	*11*	*1*	*H*	
1011 1110	*10111*	*11*	*0*	*H*	
1010 1111	*10101*	*11*	*1*	*H*	
1011 1111	*10111*	*11*	*1*	*H*	
1010 1110	*10101*	*11*	*0*	*H*	
1011 1111	*10111*	*11*	*1*	*H*	