Problem 1

Direct Mapped Cache

The accessed addresses are broken up in Table 1 based on how they would be used in a direct mapped cache for 12 bit addresses in a cache containing eight four byte cache blocks. Also shown in the last column is whether there is a hit or miss on the first run of the program, as well as noting whether the miss displaced a conflicting cache block. If a cache block is occupied but with a different tag, it is marked with "miss*" to show that there was a cache miss and the existing cache block was replaced.

Address	Binary Address	Tag	Cache Index	Byte	First Run	Subsequent Sums
200	10000_000_00	10000 (10)	000 (0)	00 (0)	miss	hit
204	10000_001_00	10000 (10)	001 (1)	00 (0)	miss	hit
208	10000_010_00	10000 (10)	010 (2)	00 (0)	miss	hit
20C	10000_011_00	10000 (10)	011 (3)	00 (0)	miss	miss*
2F4	10111_101_00	10111 (17)	101 (5)	00 (0)	miss	hit
2F0	10111_100_00	10111 (17)	100 (4)	00 (0)	miss	hit
200	10000_000_00	10000 (10)	000 (0)	00 (0)	hit	hit
204	10000_001_00	10000 (10)	001 (1)	00 (0)	hit	hit
218	10000_110_00	10000 (10)	110 (6)	00 (0)	miss	hit
21C	10000_111_00	10000 (10)	111 (7)	00 (0)	miss	hit
24C	10010_011_00	10010 (12)	011 (3)	00 (0)	miss*	miss*
2F4	10111_101_00	10111 (17)	101 (5)	00 (0)	hit	hit

Table 1: Accessed addresses broken down for direct mapped cache.

Initially, the cache is empty, shown in Table 2.

Index	V	tag	data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Table 2: Initial state

After one iteration, the cache is completely filled, shown in Table 3. There are 3 cache hits out of 12 memory accesses done.

Index	V	Tag	Data
000	Y	10000	Mem[10000000**]
001	Y	10000	Mem[10000001**]
010	Y	10000	Mem[10000011**]
011	Y	10010	Mem[10010011**]
100	Y	10111	Mem[10111100**]
101	Y	10111	Mem[10111101**]
110	Y	10000	Mem[10000110**]
111	Y	10000	Mem[10000111**]

Table 3: Direct cache after one iteration

The cache after the second iteration appear identical, as well as after the third and fourth iteration, so they are not reprinted here. However, the second iteration does have different cache performance. The only cache misses come come from the two conflicting addresses mapped to cache block 3: 20C and 24C. These both create two cache misses on iterations 2, 3, and 4.

Thus the hit rate for each iteration is 3/12, 10/12, 10/12, and 10/12. The overall hit rate is 33/48, or 68.75%.

Two Way Set Associative Cache

The breakdown of each address is shown in Table 4.

Address	Binary Address	Tag	Set Index	Byte	First Run	Subsequent Runs
200	100000_00_00	100000 (20)	00 (0)	00 (0)	miss	hit
204	100000_01_00	100000 (20)	01 (1)	00 (0)	miss	hit
208	100000_10_00	100000 (20)	10 (2)	00 (0)	miss	hit
20C	100000_11_00	100000 (20)	11 (3)	00 (0)	miss	miss*
2F4	101111_01_00	101111 (2f)	01 (1)	00 (0)	miss	hit
2F0	101111_00_00	101111 (2f)	00 (0)	00 (0)	miss	hit
200	100000_00_00	100000 (20)	00 (0)	00 (0)	hit	hit
204	100000_01_00	100000 (20)	01 (1)	00 (0)	hit	hit
218	100001_10_00	100001 (21)	10 (2)	00 (0)	miss	hit
21C	100001_11_00	100001 (21)	11 (3)	00 (0)	miss	miss*
24C	100100_11_00	100100 (24)	11 (3)	00 (0)	miss*	miss*
2F4	101111_01_00	101111 (2f)	01 (1)	00 (0)	hit	hit

Table 4: Accessed addresses broken down for two way associative cache.

The cache starts out empty, with 4 sets each containing two slots, shown in Table 5.

After one iteration, the cache is filled, shown in Table 6. There is only one conflict, when 24C is accessed.

Set Index	V_0	Tag_0	$Data_0$	V_1	Tag_1	$Data_1$
00	N			N		
01	N			N		
10	N			N		
11	N			N		

Table 5: Empty two way associative cache

Set index	V_0	Tag_0	$Data_0$	V_1	Tag_1	$Data_1$
00	Y	100000	Mem[10000000**]	Y	101111	Mem[10111100**]
01	Y	100000	Mem[10000001**]	Y	101111	Mem[10111101**]
10	Y	100000	Mem[10000010**]	Y	100001	Mem[10000110**]
11	Y	100100	Mem[10010011**]	Y	100001	Mem[10000111**]

Table 6: Two was associative cache after one iteration

For sets 0, 1, and 2, there are only two addresses that map to them so once they are warmed up, they have 100% hit rate. However, there are three addresses that map to set 3: 20C, 21C, and 24C. 20C and 21C fill up both slots in the set, then 24C replaces 20C (since it is the least recently used). At the end, the set contains 24C and 21C. On the next iteration, 20C replaces 21C (set contains 24C and 21C), then 21C replaces 24C and 24C replaces 20C, leaving the set containing 21C and 24C just like it did at the beginning of the iteration (albeit, in a different order). Thus, the contents of the cache do not change in subsequent iterations and it will not be reprinted here.

Hit rate for each iteration is 3/12, 9/12, 9/12, 9/12 and the overall hit rate is 30/48 or 62.5%.

Four way Associative Set

A four way associative set with 8 blocks will have 2 sets, each with 4 blocks. This means the 12 bit address will be split into a 9 bit key, 1 bit set index, and 2 bits of byte select. The breakdown of each address is shown in Table 7.

Initially, the cache is empty, shown in Table 8.

This time, the "Address" and "Tag" columns are filled with the addresses in hexadecimal instead of binary. By the end of the first iteration, the cache is filled, shown in Table 9. Only one conflict has occurred.

On the first run, there is only one conflict, when 24C replaces 20C. There are five addresses that map to set 1: 204, 20C, 2F4, 21C, and 24C. Just like the other two problems, the contents of the cache do not change with subsequent runs when using LRU replacement and so the cache contents for iteration 3 and 4 will be the same as in Table 9

The hit rate for each run is 3/12, 9/12, 9/12, and 9/12, for a total of 30/48 or 62.5%.

Address	Binary Address	Tag	Set Index	Byte	First Run	Subsequent Runs
200	1000000_0_00	1000000 (40)	0 (0)	00 (0)	miss	hit
204	1000000_1_00	1000000 (40)	1 (1)	00 (0)	miss	hit
208	1000001_0_00	1000001 (41)	0 (0)	00 (0)	miss	hit
20C	1000001_1_00	1000001 (41)	1 (1)	00 (0)	miss	miss*
2F4	1011110_1_00	1011110 (5e)	1 (1)	00 (0)	miss	hit
2F0	1011110_0_00	1011110 (5e)	0 (0)	00 (0)	miss	hit
200	1000000_0_00	1000000 (40)	0 (0)	00 (0)	hit	hit
204	1000000_1_00	1000000 (40)	1 (1)	00 (0)	hit	hit
218	1000011_0_00	1000011 (43)	0 (0)	00 (0)	miss	hit
21C	1000011_1_00	1000011 (43)	1 (1)	00 (0)	miss	miss*
24C	1001001_1_00	1001001 (49)	1 (1)	00 (0)	miss*	miss*
2F4	1011110_1_00	1011110 (5e)	1 (1)	00 (0)	hit	hit

Table 7: Accessed addresses broken down for four way associative cache.

Set index	V_0	Tag_0	$Data_0$	V_1	Tag_1	$Data_1$	V_2	Tag_2	$Data_2$	V_2	Tag_3	$Data_3$
0	N			N			N			N		
1	N			N			N			N		

Table 8: Four way associative cache at start.

Set index	V_0	Tag_0	$Data_0$	V_1	Tag_1	$Data_1$	V_2	Tag_2	$Data_2$	V_2	Tag_3	$Data_3$
0	Y	40	[200]	Y	41	[208]	Y	5E	[2F0]	Y	43	[218]
1	Y	40	[204]	Y	49	[24C]	Y	5E	[2F4]	Y	43	[21C]

Table 9: Four way associative cache after one iteration

Problem 2

Having larger cache blocks will reduce the number of compulsory misses. This is because when running a sequential program, more subsequent bytes are loaded for each cache fetch per miss. However, larger blocks take longer to load, resulting in more time per cache miss since the longer blocks are more expensive to load. Increasing block size without increasing the size of the cache results in fewer blocks and less granularity, causing more cache misses from replacements, especially in programs with scattered memory access.

Problem 3

The following page access sequence will be tabled for FIFO and LRU:
1 6 4 5 1 4 3 2 1 2 1 4 6 7 4 1 3 1 3 1 7

FIFO

I simulated FIFO, assuming starting state is (1 2 3 4). When a page misses, the values are shifted left to get rid of the oldest page — this is an intuitive way to tabulate the FIFO algorithm. Calculation are shown in Table 10a.

Table 10: FIFO and LRU Paging Calculations

Requested	Pages	Hit/Miss
1	(1 2 3 4)	hit
6	(2 3 4 6)	miss
4	(2 3 4 6)	hit
5	(3 4 6 5)	miss
1	(4 6 5 1)	miss
4	(4 6 5 1)	hit
3	(6 5 1 3)	miss
2	(5 1 3 2)	miss
1	(5 1 3 2)	hit
2	(5 1 3 2)	hit
1	(5 1 3 2)	hit
4	(1 3 2 4)	miss
6	(3 2 4 6)	miss
7	(2 4 6 7)	miss
4	(2 4 6 7)	hit
1	(4 6 7 1)	miss
3	(6 7 1 3)	miss
1	(6 7 1 3)	hit
3	(6 7 1 3)	hit
1	(6 7 1 3)	hit
7	(6 7 1 3)	hit

(a) FIFO paging, showing 11 hits.

Requested	Pages	Hit/Miss
1	(2 3 4 1)	hit
6	(3 4 1 6)	miss
4	(3 1 6 4)	hit
5	(1 6 4 5)	miss
1	(6 4 5 1)	hit
4	(6 5 1 4)	hit
3	(5 1 4 3)	miss
2	(1 4 3 2)	miss
1	(4 3 2 1)	hit
2	(4 3 1 2)	hit
1	(4 3 2 1)	hit
4	(3 2 1 4)	hit
6	(2 1 4 6)	miss
7	(1 4 6 7)	miss
4	(1 6 7 4)	hit
1	(6 7 4 1)	hit
3	(7 4 1 3)	miss
1	(7 4 3 1)	hit
3	(7 4 1 3)	hit
1	(7 4 3 1)	hit
7	(4 3 1 7)	hit

(b) LRU paging, showing 14 hits.

FIFO on this sequence gives 11 hits out of 21 page requests.

LRU

Now for calculating LRU, a page is moved to the bottom of the stack when it is accessed and gets a page hit, so the least recently used page floats to the top. Of course, in hardware this is not implemented using a stack but it is a helpful way to calculate LRU by hand. This process is shown in Table 10b.

LRU gives 14 hits out of 21 page requests.

Comparison

LRU performs marginally better for this problem. The lack of difference between the two is because in this sequence of page requests, hitting a page a second time doesn't mean it's likely to be hit again, so LRU and FIFO perform nearly identically.

Problem 4

A block size of 16 bytes means that 4 bits of the address go to the byte offset. Since there are 32 blocks, 5 bits is needed to address a block. The given address is 1200 (decimal). First we need to right shift off the byte offset by dividing by 4 to get 300 (decimal). Then the lower 5 bits of that by taking modulo 32 to get the answer - **12**. Block 12. To confirm this, let's write that number in binary and extract those bits.

1200 = 0b1001_01100_00

0b01100 = 12, perfect. Also, the tag will be 0b1001.

Problem 5

One way to reduce hit time is to use a simpler cache architecture — use a direct mapped cache closest to the CPU and only use more complex caches as intermediates to the main memory.

Another way is to use virtual addresses within the cache, reducing the need to do address translation during a cache hit. Of course, this means that during a cache miss, address translation must be performed. This also puts more complexity on the cache hardware that is usually handled in the main memory.

Out of order execution can hide the cache hit time delay (3-5 clock cycles) but isn't enough to deal with a cache miss (hundreds of cycles). Out of order execution and caching complement each other very nicely in this way.