1. State true or false. Add brief explanations if required (1-2 sentences):

a. Write through policies take longer write times. : True

For example, assume a 1-level cache hierarchy and a write through cache. Any change to cache data will enforce an update of the data in the main memory. This causes unnecessary delay during write operations.

b. Larger cache block sizes reduce the miss penalty.: False

Miss penalty depends on the time it takes to fetch the data from a higher entity in the memory hierarchy. Larger cache block size may reduce miss probability but will not change the miss penalty.

c. Higher associativity means higher conflict misses.: False

Higher associativity will reduce conflict misses because more associativity means more flexibility in terms of a block to fit in. In 2-way set associative cache, each block has two options to map to. In direct (1-way) chace, each block in the main memory can only ever map to one specific block in the cache.

d. Miss penalties are always lesser than hit times.: False

Miss penalty is associated with fetching data from an entity higher up in the memory hierarchy. Generally higher entities have longer fetch time. So Miss penalty is much greater than hit time.

e. The choice among different types of mapping in memory hierarchy depends on the cost of miss vs. cost of implementing associativity.: True

Associativity ensures least wastage and may reduce miss probability. But in an associative memory, the search for a data has to be parallel and this increases the hardware cost. So there is a tradeoff we must analyze during design.

2.20 (a)

Assuming/Interpreting that word size is 16 bytes.

Critical Word First: The first word fetched is the one being requested. So it will take 120 cycles to fetch it and as the data can be bypassed directly to the read port of L2 Cache, we will not consider the 4 cycle write time. So total cycles taken is 120 cycles.

Without anything: Without any optimization scheme, the entire block will be first written before the word is sent out. So it will take total time of = (120) + (3 * 16) + (4 * 4) = 184 cycles.

Early Restart: In case of early restart, we need to know the particular word access that caused the fault. There are (64 B / 16 B) = 4 Words per block.

[Considering the write not required due to bypass]

If the first word in the block was accessed then time to serve is: 120

If the second word in the block was accessed then time to serve is: 120+4+16 = 140

If the third word in the block was accessed then time to serve is: 120 + 4 + 16 + 4 + 16 = 160

If the fourth word in the block was accessed then time to serve is:

120 +4 + 16 +4 + 16 +4 +16 = 180

2.20 (b)

L2 Caches will benefit most from both critical word first and early restart because L2 cache will have greater memory fetch time. So receiving each word will take much longer than L1 cache. Hence, if critical word first and early restart are employed, then the scenario will be highly optimized.

2.21 (a)

If the sampling rate of L1 cache is same L2 cache, then each write buffer entry should be 16 bytes wide.

2.21 (b)

If each store is 64-bit then 8 Byte can be written in 2 cycles (assuming a linear relationship and given that 16 bytes take four clock cycles to write).
Now in case of a merging buffer, the update can be made only to the 8 Byte and it will take 2 cycles.
But for the non-merging buffer, the update will involve rewriting the whole 16 bytes, which will end up taking 4 cycles.
So we can expect a speed of **2x**

2.21 (c)

In case of blocking cache, the cache references which are not associated with the missing data are also blocked.
In case of non-blocking cache, a queue is used to put in the fetch due to miss request while the cache continues to serve.
Incase of non blocking cache, buffer data can be fetched even during a cache miss. For blocking cache, the effectiveness of buffer is minimized.

2.22

Based on the information provided the average time spent accessing the cache hierarchy are as follows:

(a) 1000 + (100 * 16) + (10*200) = 4600 cc / 1k Inst
(b) 1000 + (100 * 4) + (50 *16) + (10*200) = 4200 cc / 1k Inst
(c) 1000 + (100 * 2) + (80 * 8) + (40 *16) + (10 * 200) = 4480 cc / 1k Inst

We observe that if the memory hierarchy is:

Too Shallow: Big time hit from each miss.

Too Depp: Too much time wasted querying all the memory entities on the way.

2.22 (Alternative Interpretation)

If the amount of miss at each cache entity is to be computed based on the number of instructions that reaches it then the following calculation holds:

(a) 1000 + (100 * 16) + (1 * 200) = 2800 cc / 1k Inst
(b) 1000 + (100 * 4) + (5 * 16) + (0.05 * 200) = 1490 cc / 1k Inst
(c) 1000 + (100 * 2) + (8 * 8) + ($\frac{8}{25}$ * 16) + ($\frac{8}{2500}$ * 200) = 1269.76 cc /1k Inst

Too Shallow: Big time hit from each miss.

Too Depp: Too much time wasted querying all the memory entities on the way.

B.1(a)

Given the information, Average memory access time is = $\frac{100 + (3 * 110)}{100}$ = 4.3 cycles

B.1(b)

The array size is 1GB. Data cache size is 64KB. So the fraction of array that can reside at a time in the cache is = $\frac{64KB}{1GB} = 2^{-14}$ = Probability of hit in random access scenario.

So the miss rate is = $1 - 2^{-14}$

So the average memory access time = $\frac{100 + (1-2^{-14}) * 100 * 110}{100}$

$$= \frac{100 + (0.99993)*100*110}{100}$$

= 110.9923 cycles

B.1(c)

When cache is disabled the average memory access time = $\frac{100 * 105}{100}$ = 105 cycles

The entire idea of cache was built around the assumption that, subsequent memory access will be in close proximity of each other. Also the same memory location may be accessed multiple times in a short amount of time. All these assumptions are destroyed the moment we do random access of the memory. The random access was emulated approximately by using the 1GB array. Due to extremely high miss rate, having a cache hurts more than it helps.

B.1(d)

Assuming that **m** = the miss rate threshold after which the effectiveness of cache is removed.

Then we can find m by solving the following equation representing the average access time in-presence and in- absence of the cache.

$$\frac{100 + (m * 100 * 110)}{100} = \frac{105 * 100}{100}$$

=> 1 + 110m = 105

=> m = $\frac{104}{110}$ = 0.95

So any more than 0.95 miss rate will render the cache useless and will start to hurt more than help.

B.2(a)

If the cache is fully associative then all the main memory blocks can map to any block in the cache.

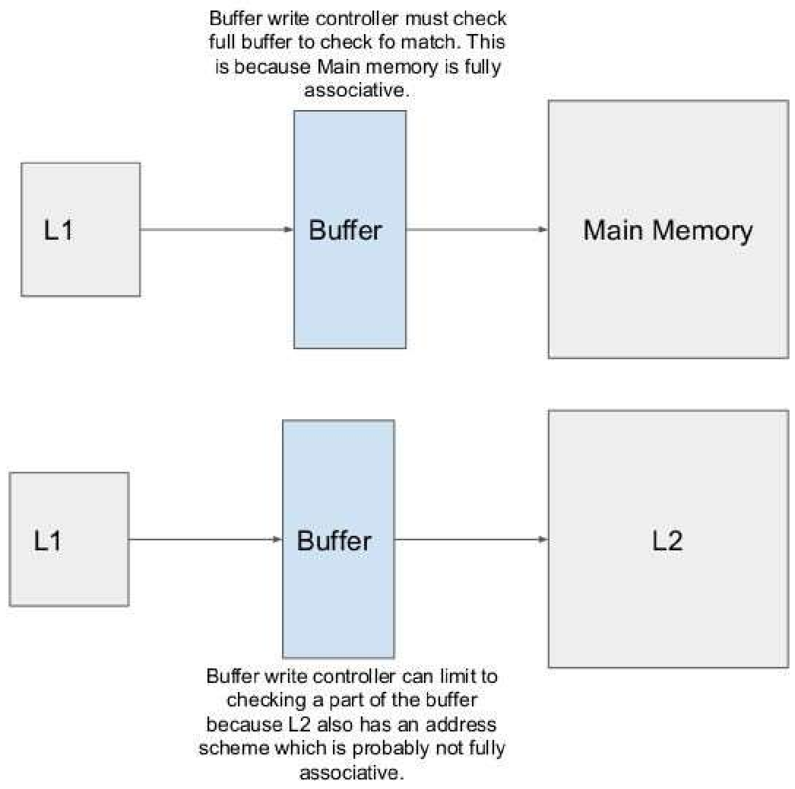
Cache Block	Set	Way	Possible Memory Blocks
0	0	All-Way	M0, M1, …., M31
1	0	All-Way	M0, M1, …., M31
2	0	All-Way	M0, M1, …., M31
3	0	All-Way	M0, M1, …., M31
4	0	All-Way	M0, M1, …., M31
5	0	All-Way	M0, M1, …., M31
6	0	All-Way	M0, M1, …., M31
7	0	All-Way	M0, M1, …., M31

B.2(b)

For a 4-way set associative cache, we will have 2 sets.

Cache Block	Set	Way	Possible Memory Blocks
0	0	4-Way	M0, M2, M4, …., M30
1	0	4-Way	M0, M2, M4, …., M30
2	0	4-Way	M0, M2, M4, …., M30
3	0	4-Way	M0, M2, M4, …., M30
4	1	4-Way	M1, M3, M5, ….,M31
5	1	4-Way	M1, M3, M5, ….,M31
6	1	4-Way	M1, M3, M5, ….,M31
7	1	4-Way	M1, M3, M5, ….,M31

B.7


Buffer write controller must check full buffer to check fo match. This is because Main memory is fully associative.
L1
Buffer
Main Memory
L1
Buffer
L2
Buffer write controller can limit to checking a part of the buffer because L2 also has an address scheme which is probably not fully associative.

Since the main memory fully associative the complete buffer must be checked before carrying out an update. This is will require more hardware to implement. In contrast, we can easily implement the same scheme with the buffer between an L1 and L2 cache, just by checking a subset of buffer. This is because L2 is probably not fully associative and can less hardware will be required for checking the buffer content’s address.