CSM 61C — Caches

Spring 2020 — Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Spring 2018 Midterm 2 Q5
- Summer 2015 MT2 Q5
- Summer 2018 Final Q6
- Fall 2015 Final MT2-4
- Fall 2017 Midterm 2 Q5
- Fall 2018 Midterm Q3
- Summer 2018 Midterm 2 Q5
- Summer 2018 Final Q1
- Fall 2019 Final Q8
- Summer 2018 Final Q8

Problem 5 ***$$$*** **(17 points)**

You are given following RISC-V Code:

```
// a0:  Integer array location
// a1:  End bound of the array
// a2:  Offset to new location in words
// Assume these registers hold the following values
// at the start of the program:
// a0 -> 0x1000, a1 -> 2048, a2 -> 2048

    add t0, x0, x0
    slli t3, a2, 2
loop:
    beq t0, a1, done
    slli t1, t0, 2
    add t1, t1, a0
    lw t2, 0(t1)
    add t1, t1, t3
    sw t2, 0(t1)
    addi t0, t0, 4
    jal x0, loop
done:
    ...
```

Solution: Equivalent C Code:

```
int a[512]; // At Address 0x1000
...
int b[512]; // At Address 0x1000 + 2048 × 4
for (int i = 0; i < 2048; i += 4) {
    b[i] = a[i];
}
```

Assume there is enough memory allocated for the array such that there are no memory out of bounds issues. Also assume the code is run on a machine with a 32-bit address space. Questions will only involve the code starting from `loop` and only refer to one data cache. This cache uses a LRU replacement policy and is write allocate unless otherwise stated.

(a) Consider an 8 words/block, 512B **direct-mapped** data cache. The cache starts empty and we run the program above to completion.

(i) Calculate the number of tag, index, and offset bits for this cache.

Solution: Tag: 23 Index: 4 Offset: 5
8 words/block under a 32-bit address space means $4 \times 8 = 2^5$ bytes/block.
As a result, this also means there are $512/(2^5) = 16 = 2^4$ blocks total.
We need a 32-bit address, so the tag bits must be the remaining 23.

(ii) What is the hit rate of this direct-mapped cache?

Solution: 0
In lw t2, 0(t1), t1 at first is just 0 + 0x1000, or 0b1000 0000 00000, so our cache pulls in 0x1000 to 0x101F into Index 0x0.
For the corresponding sw, t1 is the old t1 plus the offset (2048×4), or 0b11000 0000 00000. This maps to the same Index as before, so the previous block is evicted.
The next iteration's lw looks for 4×4 + 0x1000, or 0b1000 0000 10000, which maps to Index 0x0. It evicts the sw's block to make way for its own.
This ping-pong of evictions continues, so the cache never returns a hit.

(iii) What types of of cache misses occur? Mark all that apply.

○ Capacity [✓ ● Conflict]

[✓ ● Compulsory]

Solution: From (ii), we see that there are Compulsory misses if the program hits an array location it has never accessed before. The cycle of evictions is a result of Conflict misses, where the different Tags+matching Indices situation means previously-in-cache information is lost then re-accessed. There are no Capacity misses; adding more blocks won't solve this situation (and our current situation works as-if there was always just one-block).

(iv) Assume the cache is emptied and we re-run the program above to completion, but this time with a cache **block size of 4 words**. What is the hit rate of this new cache?

Solution: 0
No difference. Due to the offset between the two arrays, only the Tag differs between the two array addresses.

(b) Now consider an 8 words/block, 512B **Two-Way Set Associative** data cache. The cache starts empty and we run the program above to completion.

(i) What is the hit rate of this Two-Way Set Associative cache?

Solution: 1/2
We can handle the Tag-difference situation, but what now is important is our stride, or how i changes. A strides of 4 results in lookups like:
(lw) 0b01000 0000 00000
(sw) 0b11000 0000 00000
(lw) 0b01000 0000 10000
(sw) 0b11000 0000 10000
(lw) 0b01000 0001 00000
For each Tag-Index pair, we only see two possible accesses, so the cache helps get the second access faster. (Cache pulls entire block in)

(ii) What types of of cache misses occur? Mark all that apply.

○ Capacity ○ Conflict

● Compulsory

Solution: From (i), there are Compulsory misses if the program hits an array location it has never accessed before (e.g. new Tag-Index). Unlike (a), the lw and sw operations don't Conflict in the cache. There continues to be no Capacity misses; adding more blocks won't help our cache.

(iii) Assume the cache is emptied and we re-run the program above to completion, but this time with a cache **block size of 4 words**. What is the hit rate of this new cache?

Solution: 0
4 words/block = 16 bytes/block (2^4)
512/(2^4) = 32 blocks total (2^5)
Changing block size shifts our T:I:O allocations, so our accesses look like:
(lw) 0b01000 00000 0000
(sw) 0b11000 00000 0000
(lw) 0b01000 00001 0000
(sw) 0b11000 00001 0000
As seen, Tag-Index pairs are now unique; they never repeat. As such, the information in cache is never used, so there can be no hits.

(c) Now consider a 8 words/block, 512B **Four-Way Set Associative** data cache. The cache starts empty and we run the program above to completion.

(i) What is the hit rate of this Four-Way Set Associative cache?

Solution: 1/2
As seen from the memory access pattern in (b)(i), Tag-Index pairs come in two. Increasing Associativity does not change this, so making that cache Four-Way doesn't change the hit rate.

(ii) Assume the Four-Way Set Associative cache is emptied and we re-run the program above to completion, but this time with a **random replacement policy**. How would the hit rate most likely change compared to part c.i?

○ Increase ○ Stay the Same

[✓ Decrease]

Solution: The best replacement policy would be the one where the page evicted is the one that will not be accessed for the longest time. LRU is a better approximation of this strategy that Random Replacement, given that generally programs have some form of locality. The worst case for Random Replacement here is if we evict the block we need later, a situation in this scenario can't happen in LRU:

Assuming the Cache has one block left
(lw) 0b01000 0000 00000 # Block granted to 0b01000
(sw) 0b11000 0000 00000 # Cache full; Randomly Evict 0b01000
(lw) 0b01000 0000 10000 # Cache full; Randomly Evict 0b11000
(sw) 0b11000 0000 10000 # Cache full; Randomly Evict 0b01000
(lw) 0b01000 0001 00000

(d) Consider the 8 words/block, 512B **direct-mapped** data cache again. The cache starts empty and we run the program above to completion, **except this time with a2 initialized to 2056**. What is the hit rate of this direct-mapped cache?

Solution: 1/2
The offset is now 2056 × 4, or 0b10000 0001 00000. As a result, the lw/sw now don't share the same index location, so we don't have the eviction ping pong from (a).
(lw) 0b01000 0000 00000 Compulsory
(sw) 0b11000 0001 00000 Compulsory, but doesn't affect earlier lw info
(lw) 0b01000 0000 10000 Hit
(sw) 0b11000 0001 10000 Hit
(lw) 0b01000 0001 00000 Compulsory

Q5) Caches (17 pts)

Assume we are working in a 32-bit physical address space. We have two possible data caches: cache X is a direct-mapped cache, while cache Y is 2-way associative with LRU replacement policy. Both are 4 KiB caches with 512 B blocks and use write-back and write-allocate policies.

a) Calculate the number of bits used for Tag, Index and Offset:

Cache	Tag bits	Index bits	Offset bits
X	20	3	9
Y	21	2	9

Use the code below to answer the following parts. Assume that ints are 4 B and doubles are 8 B.

```
//2^11 elements in the double array
int DOUBLE_ARRAY_SIZE = 2 * 1024;                //2^14 bytes
double double_arr[DOUBLE_ARRAY_SIZE];

for (int i = 0; i < DOUBLE_ARRAY_SIZE; i++) /* loop 1 */
      double_arr[i] = i;
for (int i = 0; i < DOUBLE_ARRAY_SIZE;  i += 8) /* loop 2 */
     double_arr[i] *= double_arr[0];
```

b) What is the hit rate for each cache if we run only loop 1? (hint: they're both the same). What types of misses do we get?

Both have a hit rate of 63/64. Compulsory

c) What is the hit rate of each cache when you execute loop 2? Assume that you have executed loop 1. Assume the worst case ordering of accesses within a single iteration of the loop if multiple orders are possible. You may leave your answer as an expression involving products and sums of fractions.

X: ____See Below______________________ Y: ________See Below________________

We are accessing with a stride of 8*8B = 64B while our block size is 512B. Thus, we have 8 accesses in each block. Since the array size is 2Ki*8B = 16KiB and our caches are 4KiB, the actual data is 4 times the size of our caches.

In cache X, the first quarter of the array will have a hit rate of 23/24 since we only encounter misses in each new block. In the rest of the array, however, we get a ping-pong effect on the first block as the loop requires double_arr[0]. For the 2nd-4th quarter of the array, the first block accesses will be (double_arr[i], double_arr[0], double_arr[i]):
Access 1: M, M, M
Access 2: H, M, M
Access 3: H, M, M; then H, M, M until access 8.
This yields a hit rate of 7/24. The second through eighth block accesses have 23/24 hit rate.

Putting it all together, since we are accessing 4*8 blocks in total,
Hit rate = 29/32 * 23/24 + 3/32 * 7/24

Cache Y has a hit rate of 23/24 because there is no ping-pong effect in the first block.

d) Compute the AMAT for the following system with 3 levels of caches. (You should not need any information from the previous parts of this problem.) Give your answer as a decimal value.

L1$	**L2$**	**L3$**	**Main Memory**
Global miss rate: 50% Hit time: 1ns	Local miss rate: 20% Hit time: 5ns	Local miss rate: 1% Hit time: 15ns	Hit time: 500ns

1 + .5*(5 + .20*(15 + .01*(500))) = 5.5 ns

Summer 2018 Final Solution

SID: ________________

Question 6: Cache These Hands (16 pts)

A machine with a 19 bit address space has a single 256 B cache. The cache is 4-way set associative with 8 total entries.

1. Determine the number of bits in Tag, Index, and Offset fields for an address on this machine.

Tag: 13 Index: 1 Offset: 5

The following piece of code is executed on the aforementioned machine. This code computes an outer product of a **N x 1** vector A and a **1 x N** vector B, placing the result in a **N x N** matrix C. Use this code to answer the follow questions about the hit rate the code was produced.

For all questions assume the following:

- sizeof (double) == 8
- A = 0x10000
- B = 0x20000
- C = 0x30000
- The cache begins cold before each question.
- Code is executed from left to right.

```
#define N 16

void outer_product (double *A, double *B, double *C) {
      for (int i = 0; i < N; i++) {
                for (int j` = 0; j < N; j++) {
                          C [j + i * N] = A[ i ] * B[ j ];
                }
      }
}
```

SID: _______________

2. What is the hit rate for executing this code if it uses LRU replacement and is a write back cache with write allocate on a miss? Fill in all blanks for credit.

HR for accesses to A: **63/64**

HR for accesses to B: **27/32**

HR for accesses to C: **3/4**

OVERALL HR: **⅓ * 63/64 + ⅓ * 27/32 + ⅓ * ¾ = 21 / 64 + 9 / 32 + 1/4 = 55/64**

Explanation: We start by attempting to find a pattern when i = 0. Accesses occur left to right so first A is read, then B, and then C. On first iteration all 3 elements will miss and all 3 blocks will be loaded into the cache because the cache is 4 way set associative. Since each block is 32 Bytes in size and each access is 8 Bytes (1 double), the all access will hit giving this result:
i = 0, j = 0
M M M
H H H
H H H
H H H

where the first access is always A, the second B and the third C. Then when j = 4, B and C will move to a new block but A is still in the cache, so it will hit. Otherwise the pattern is the same

i = 0, j = 4
H M M
H H H
H H H
H H H

Then when j = 8, B and C once again enter set 0. B will fit but then that set will be full. Since A keeps being accessed, B's old value will be evicted. So the pattern will remain for j = 8. However when j = 12 B's old value will not be evicted.

SID: ________________

i = 0, j = 8
H M M
H H H
H H H
H H H

i = 0, j = 12

When i = 1 only two notable changes occur. A will still be loaded in, because this will not be a miss again until i is divisible by 4 and B[4] and B[12] will still be in the cache.

This means that from i = 1 to i = 3

A will not miss, B will miss only twice per iteration (6 times total) and C will maintain its current hit rate. When i = 4 then the pattern will repeat, although it switches sets (B[0] and B[8] will remain in the cache starting from i = 5). As a result the overall hit rate for each part is:

A has 1 miss, so 63/64
B has 10 misses so 54/64 = 27/32
C misses one in every 4 access, so 3/4

3. What is the hit rate for executing this code if it uses LRU replacement and is a write back cache with no write allocate on a miss? Fill in all blanks for credit.

HR for accesses to A: 63/64

HR for accesses to B: 63/64

HR for accesses to C: 0

OVERALL HR: 63/64

SID: ____________

Explanation: There are many ways to do this question. The easiest barely looks at the access patterns. Since C is a different address from A and B, no part of C will ever enter the cache. The total size of the cache is 256 B, A contains 128 B and B contains 128 B so A and B both fill the cache. Each element of A and each element of B is accessed 16 times, so each Vector is accessed 256 times with only 4 compulsory misses.

As a result,

A = 252/256 = 63/64
B = 252/256 = 63/64
C = 0

MT2-4: If you do well, it's *clobbering* time! (12 points)

The information for one student in regards to clobbering a single midterm is captured in the data of the following *tightly-packed* struct:

```
typedef struct student {
    int studentID;
    float oldZScore;
    float newZScore;
    int clobber;          /* a value equal to 1 if a student clobbers,
                             0 if otherwise */
} student;
```

We run the following code on a 32-bit machine with a 4 KiB write-back cache. `importStudent()` returns a `struct student` that is in the course roster and that has not been returned by `importStudent()` previously. For simplicity, assume `importStudent()` does not affect the cache.

```
int ARR_SIZE = 512; //Class size rounded down for simplicity
student *61CStudents = (student *) malloc (sizeof(student) * ARR_SIZE);

/* Assume malloc returns a cache block aligned address */
for (int i = 0; i < ARR_SIZE; i++) {                      <=== part I
    61CStudents[i] = importStudent()  <- what does import student do?
}

for (int i = 0; i < ARR_SIZE; i++) {                      <=== part II
    if (61CStudents[i].oldZscore > 61CStudents[i].newZscore){
      61CStudents[i].clobber = 0;
    } else {




      61CStudents[i].clobber = 1;
    }
}
```

a. How many bytes is needed to store the information for a single student?
The struct has 4 fields, each of 4 bytes, so 16 bytes

b. Assume that the block size is 32 B. What is the tag:index:offset breakdown of the cache?
The final clarifications told the students to assume a direct mapped cache. Thus, if the cache size is 4KiB (2^12), and the block size is 32 B (2^5), there must be 2^12/2^5 = 2^7 blocks. Thus, there are 7 index bits, 5 offset bits, and 20 tag bits. 20:7:5

c. At the label part I, assume that 61CStudents is filled with the correct data. What type of misses will occur from memory accesses during the process? Why?

The misses that will occur from executing the for loop at label part I will be compulsory misses, because the loop will be accessing student structs that will be accessed for the first time.

d. Suppose we run the code again and the cache block size is now 8 B long and the cache is direct mapped. For the for-loop in part II, what is the miss rate in the best case scenario (we want the highest hit rate possible)? What type of misses occur?
The if-else block in part II has 3 memory accesses. Since half of the struct fits in a block of the cache, 61CStudents[i].oldZscore would be a miss and load the first two elements of the struct into a block, 61CStudents[i].newZscore would be a miss and load the last two elements of the struct into a block, and 61CStudents[i].clobber would be a hit in the second block loaded in. Thus, the miss rate is 2/3. These misses are capacity misses because even if the cache was fully associative, the array of structs does not fit in this cache, and entries would have to be evicted due to the cache size in the fully associative case.

e. For the for loop in part II, assume that the cache block size is now 128B.

i. If the cache is direct-mapped, what is the hit rate?
8 students per block. 3 memory access per student, 1 miss and 23 hits, so 23/24

ii. If the cache is fully associative, what is the hit rate? Does associativity help? Why or why not?
It would still be 23/24, because the array is being accessed sequentially, so associativity makes no difference.

Q5: Do the Monster Cache

a) For the following cache questions, please **bubble** in your answer on the answer sheet:

I. True or False? A fully associative cache has no conflict misses.

II. True or False? A write-back cache must write to memory *immediately* when a block is modified.

For all portions of this question assume that an integer is one word in size and that ALL operations occur from left to right. Consider a 16-way set-associative cache with two-word blocks, 16 sets and a 128 TiB physical byte-addressed address space.

b) When breaking down a physical address into the Tag, Index, and Offset fields, how many bits long is each field? (i.e. what is the T:I:O for the cache?) Write your answer on the blanks provided on your answer sheet.

T: 40 I: 4 O: 3

Total address bits = $\log_2(128Ti) = \log_2(128*2^{40}) = 47$

offset bits = $\log_2$(2 words) = $\log_2$(8 bytes) = 3

index = $\log_2$(16 sets) = 4

Tag = 47 - 4 - 3 = 40

Now consider the following code segment:

```
void sequence(int* A, int* B) {
    int i;
    //PART C
    for (i = 0; i < 16; i++) {
      B[i] = 2;
      A[i] = 4;
    }
    //PARTS D & E
    for (i = 16; i < 272; i++) {
      B[i] = B[i - 8] + A[i - 8];
      A[i] = B[i - 16] + A[i - 16];
    }
}
```

Let A's address for the following code segment be 0x10000 and B's address be 0x20000. Assume that an integer is one word in size, that ALL OPERATIONS are evaluated from left to right, and that all of the cache's valid bits are set to zero before running the sequence function. Remember to write all of your answers to the questions below on your answer sheet.

c) What is the hit rate for running the loop below **PART C** using the cache from (b)?

[✓ 1/2] 0 7/8 3/4 15/16

Miss/Hit Pattern is MMHH.
The hit rate is 50%.

d) What is the cache hit rate for cache accesses that occur below **PARTS D & E** when running sequence after **PART C** completes.

5/6
The miss/hit pattern is HHMHHM, HHHHHH.
Each 6 accesses are from one iteration of the for loop.
Cache hit rate is 0.5*4/6 + 0.5 * 6/6 = 5/6

e) Assuming the cache and block sizes are held constant, what is the minimum cache associativity that results in a maximum hit rate for the segment of sequence?

1-way	[✓ 2-way]	4-way
8-way	16-way	32-way

The minimum associativity is 2 ways because we have two different arrays A and B and we would to avoid conflict among accesses to them.

f) Assume that sequence ran above resulted in a total of 50 accesses (this may or may not be true) and that it was run on a computer with an L1 and L2 cache. Say that the L1 cache has an access time of 10ns, the L2 cache has an access time of 20ns, main memory has an access time of 50ns, the L1 cache has an 80% hit rate, and that the total AMAT for running sequence is 16 ns. What is the local hit rate for the L2 cache? You do not need to know either of the caches' parameters for this question. **Please write your answer as a decimal, up to 2 decimal places, on your answer sheet.**

10 + 20%(20 + L2localMissrate*50) = 16
10 + 4 + L2localMissrate*10 = 16
L2 local missrate = (16-14)/10 = 0.2
L2 local hit rate = 1- 0.2 = 0.8

Fall 2018 Midterm Solution

Q3) Cache money, dollar bills, y'all. (18 points)

We have a 32-bit machine, and a 4 KiB direct mapped cache with 256B blocks. We run the following code from scratch, with the cache initially cold.

`uint8_t addup() {`
`    uint8_t A[1024], sum = 0; // 8-bit unsigned`
`    touch(A);`
`    for (int i = 0; i < 1024; i++) { sum += A[i]; }`
`    return(sum-1);`
`}`
	`void touch(uint8_t *A) {`
	`    // Touch random location`
	`    // in A between`
	`    // A[0] to A[1023], inclusive`
	`    A[random(0,1024)] = 0;`
	`}`

a) Assume **sum** has the smallest possible value after the loop. What would **addup** return? ___255___
The smallest value is 0, when you subtract 1 you have "negative overflow" back to the biggest representable number of an 8-bit unsigned value, which is **255**.

b) Let **A = 0x100061C0**. What cache index is **A[0]**? ___1___
We tried to design this cache so the numbers are easy. 256B blocks is 8 bits for the offset, or the last two nibbles. 4KiB bytes in cache / 256B bytes/block = 16 blocks or 4 bits there, which is the 3rd nibble from the right. So the index is **1**.

c) Let **A = 0x100061C0**. If the cache has a hit at **i=0** in the loop, what is the maximum value returned by **random**? ___63___
If we look at the 8 bits of offset, the value if C0, which is 0b1100 0000, (¼ of the way from the left since the top two bits are 11) so any value of random between 0 and 0b11 1111 would leave us in that block (before the 8 bit offset bubbles over, and another index is randomly touched), and 0b11 1111 is **63**.

For d and e, assume we don't know where **A** is.

d) What's the fewest misses caused by the loop? ___3___
For the fewest misses, let's assume A is block aligned. There are only 4 blocks ever used by the cache in this case, and touch gives us one free hit, so that's **3 compulsory misses**.

e) What's the most misses caused by the loop? ___4___
For the most misses, let's assume A is not block aligned. There are only 5 blocks ever used by the cache in this case, and touch gives us one free hit, so that's **4 compulsory misses**.

f) If we change to a fully associative LRU cache, how would c, d, e's values change? (select 1 per box)
The array is well smaller than the array, nothing gets kicked out, so **nothing would change**

c: ○up ○down [✓ same]	d: ○up ○down [✓ same]	e: ○up ○down [✓ same]

g) When evaluating your code's performance, you find an AMAT of 4 cycles. Your L1 cache hits in 2 cycles and it takes 100 cycles to go to main memory. What is the *L1 hit rate*? ___98___%
AMAT = L1 Hit time + L1 Miss rate + L1 Miss penalty, and L1 hit rate = 1 - L1 miss rate
4 = 2 + L1MR * 100 ⇒ L1MR = 1/50 = 2%, so **L1 hit rate = 98%**

Q4) *RISC-V business: I'm in a CS61C midterm & I'm being chased by Guido the killer pimp...* (14 points)

a) Write a function in RISC-V code to return *0 if the input 32-bit float = ∞*, else a non-zero value. The input and output will be stored in **a0**, as usual.
(If you use 2 lines=3pts. 3 lines=2 pts)

```
isNotInfinity: lui a1, 0x7F800
               xor a0, a0, a1
               ret
```

SID: _______________

Question 5: C.R.E.A.M. (15 pts)

1) A machine has an 8 way set associative cache with 512 B of data. The size of each block is 16 B and there are 8 MiB of main memory. How large are the tag, index, and offset fields for an address on this machine using this cache?

Tag: 17 Index: 2 Offset: 4

2) Now we have a different machine with two caches, an L1 and an L2 cache. Both caches are direct mapped caches. The L1 cache can hold 256 B of data and the L2 cache can hold 4 KiB of data. Assume the following code is run on this machine:

```
#define ARR_SIZE 2048

uint16_t sum (uint16_t *arr) {
    total = 0;
    for (int i = 0; i < ARR_SIZE; i++) {
       total += arr[i];
    }
    return total;
}
```

This produces a hit rate (HR) of 7/8 for the L1 cache and 3/4 for the L2 cache. Given that `arr` is a block aligned address and `sizeof (uint16_t) == 2`:

A. What is the blocksize of the L1 cache **in bytes** that produces its hit rate?

L1_BLOCKSIZE: 16 B

B. Use the variable **Y** to represent the answer to part A. What is the blocksize of the L2 cache **in bytes** that produces its hit rate? Express your answer as a function of **Y** and NOT as a single number.

L2_BLOCKSIZE: 4 * Y (64 B)

Recall that the L1 HR is 7/8 and the L2 HR is 3/4. If the L1 Cache has a hit time of 2 cycles, the L2 cache has a hit time of 8 cycles, and main memory has a hit time of 96 cycles:

3) How many total cycles are spent accessing memory on this piece of code? Express your answer in the form **C * 2^i**, where **C** is an integer not divisible by 2.

TOTAL_CYCLES: 2048 * (2 + 1/8 * (8 + 1/4 * 96)) = 2^11 * (2 + 8/8 + 96/32) = 2^11 * (6) = 3 * 2^12

4) If we change the L1 cache from being direct mapped to being fully associative with LRU, how does its HR change on the same code?

Ⓐ Increases Ⓑ Decreases [✓ Ⓒ No Change] Ⓓ Impossible to tell

Fall 2019 Final Solution

Q8) This is for all the money! (15 pts = 3 + 7 + 5)

Assume we have a single-level, 1 KiB direct-mapped L1 cache with 16-byte blocks. We have 4 GiB of memory. An integer is 4 bytes. The array is block-aligned.

```
#define LEN 2048

int ARRAY[LEN];
int main() {
    for (int i = 0; i < LEN - 256; i+=256) {
        ARRAY[i] = ARRAY[i] + ARRAY[i+1] + ARRAY[i+256];
        ARRAY[i] += 10;
    }
}
```

a) Calculate the number of tag, index, and offset **bits** in the L1 cache.

T:22	I:6	O:4

SHOW YOUR WORK

Offset: log2(block size) = log2(16) = 4
Index: log2(cache size / block size) = log2(1 KiB / 16) = log2(64) = 6
Tag:
First find total address bits log2(4 GiB) = log2(4 * 2^30) = log2(2^32) = 32
Then 32 - Index - Offset = 32 - 6 - 4 = 22

b) What is the hit rate for the code above? Assume C processes expressions left-to-right.

50%

SHOW YOUR WORK

Every iteration it's
ARRAY[i] read MISS
ARRAY[i+1] read HIT
ARRAY[i+256] read CONFLICT→ MISS
ARRAY[i] write CONFLICT→ MISS
ARRAY[i] read HIT
ARRAY[i] write HIT
3 MISSES, 3 HITS. 50% hit rate.

c) You decide to add an L2 cache to your system! You shrink your L1 cache, so it now takes 3 cycles to access L1. Since you have a larger L2 cache, it takes 50 cycles to access L2. The L1 cache has a hit rate of 25% while the L2 cache has a hit rate of 90%. It takes 500 cycles to access physical memory. What is the average memory access time in cycles?

78

SHOW YOUR WORK

AMAT = 3 + ¾ (50 + 1/10 500)
AMAT = 3 + ¾ (50 + 50)
AMAT = 3 + ¾ (100)
AMAT = 3 + 75
AMAT = 78

SID: _______________

Question 8: Mr. MOESI (6 pts)

For this question you will be asked to determine which cache coherence scheme(s) can fulfill tasks efficiently based upon assumptions of what task our machine must perform. For each question there will be two parts:

- **Expected Behavior**: the behavior that your scheme must perform efficiently.
- **Necessary Behavior**: the behavior that doesn't need to be performed efficiently but must be supported.

You should select **all** schemes that can handle the expected behavior with maximum efficiency.

For all scenarios, we have the following assumptions:

- The machine has multiple cores
- **Writing** to memory is very very slow (a performance bottleneck)

If this is unclear, consider an example. If the expected behavior involves writing and **MSI** and **MOESI** can do fewer writes to memory than **MESI**, then you would select **MSI** and **MOESI** and not **MESI**.

1. **Expected Behavior:** Processing completely read only data.
   Necessary Behavior: Nothing additional.

   [✓ Ⓐ **MSI**] [✓ Ⓑ **MESI**] [✓ Ⓒ **MOESI**]

2. **Expected Behavior:** A single program with threads for different purposes. A single thread will write in the information about a user. Then after this write, the other three threads will in parallel perform different computations based upon the user's data (each written to different, unique location).
   Necessary Behavior: Writing to a final shared location may be necessary to accumulate results once all threads have completed.

   Ⓐ **MSI** Ⓑ **MESI** [✓ Ⓒ **MOESI**]

3. **Expected Behavior:** Processing a unique program on each core (with its own memory space) and quick reading from memory shared by programs.
   Necessary Behavior: Writing to data that is shared across programs.

   Ⓐ **MSI** [✓ Ⓑ **MESI**] [✓ Ⓒ **MOESI**]