Started on	Monday, 9 March 2020, 4:50 PM
State	Finished
Completed on	Monday, 9 March 2020, 5:41 PM
Time taken	50 mins 5 secs
Marks	30.24/43.00
Grade	**7.03** out of 10.00 (**70**%)

Question **1**

Incorrect

Mark 0.00 out of 2.00

Estimate the time (in ms) to access a sector on the following disk

Rotational Rate	Average Seek Time	Average Sectors per track
7500	8 ms	495

You may use an expression if that's useful.

16

Your last answer was interpreted as follows: 16

Incorrect answer.
That's either wrong or too far off the precise value.

The total access time is is $T_{\text{access}} = T_{\text{avg. seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$.

The seek time is $T_{\text{seek}} = 8$, as specified in the problem.

The rotation delay is
$T_{\text{avg rotation}} = 1/2 \times T_{\text{max rotation}} = 1/2 * (60/7500) * 1000\text{ms/s}$.

The transfer delay is
$T_{\text{avg transfer}} = (60/7500) * (1/7500)\text{sectors/track} * 1000\text{ms/s}$.

The total access time is thus the sum or $\frac{5948}{495}$ = 12.0161616162.

A correct answer is $\frac{5948}{495}$, which can be typed in as follows: `5948/495`

Question **2**

Correct

Mark 4.00 out of 4.00

At a high level, you can model the time to access a hard disk drive as the "access time" and the "transfer time" and a solid state disk can be modeled in a similar way.

Assume that a specific hard disk drive has an average access time of 15ms (*i.e.* the seek and rotational delay sums to 15ms) and a throughput or transfer rate of 170MBytes/s, where a megabyte is measured as 1024^2.

A solid-state drive (SSD) has an average access time of 0.037ms and a throughput of 410MB/s -- the access time serves the same role as the combined "seek" and "rotationaal" delay of a disk and represents the time to talk to the SSD and the overhead time for the non-volatile memory to be accessed.

Big Reads

Some applications, such as playing back a movie, read large files -- they do a single "seek" or access to the beginning of the file and then read a large collection of blocks. Assume that a movie playing application does a single "access" and then reads a 165MB file.

How many "movies per second" can be processed by the hard disk drive?

1.014

Your last answer was interpreted as follows: 1.014

Correct answer, well done.
Close enough!

How many "movies per second" can be processed by the SSD disk drive?

2.485

Your last answer was interpreted as follows: 2.485

Correct answer, well done.
Close enough!

Each answer should be accurate to within 5% "movies per second" and you can use algebraic expressions if you like.

Random I/O

Many other applications are limited by "IOPS", or the "random I/O operations per second". An example of this would be a database that needs to seek to a part of the disk and read a small amount of data of 512 bytes repeatedly.

How many "IOPS" can be processed by the hard disk drive?

66.6539

Your last answer was interpreted as follows: 66.6539

Correct answer, well done.
Close enough!

How many "IOPS" can be processed by the SSD disk drive?

26184.2276

Your last answer was interpreted as follows: 26184.2276

Correct answer, well done.
Close enough!

Each answer should be accurate to within one "IOPS" and you can use algebraic expressions if you like.

The time to seek to and read a 165MB file is $T_{\text{fs}} = T_{\text{seek}} + (165/\text{throughput})$. The speed in files per second is $\frac{1}{T_{\text{fs}}}$.

For the disk, this is $\frac{1}{\frac{15}{1000}+\frac{165}{170}} = \frac{3400}{3351} = 1.01462250075$.

For the ssd, this is $\frac{1}{\frac{\frac{37}{1000}}{1000}+\frac{165}{410}} = \frac{41000000}{16501517} = 2.48462005039$.

There's a relatively small (3-10x) difference in the number of movies-per-second that can be served, related to the difference in throughput.

The number of IOPS is $1/T_{\text{seek}}$.

For the disk this is $\frac{1io}{15ms*\frac{1s}{1000ms}} = \frac{200}{3} = 66.6666666667$.

For the ssd this is $\frac{1io}{0.037ms*\frac{1s}{1000ms}} = \frac{1000000}{37} = 27027.027027$.

There's a relatively large (~500x) difference in the number of movies-per-second that can be served, related to the difference in latency.

A correct answer is $\frac{3400}{3351}$, which can be typed in as follows: `3400/3351`

A correct answer is $\frac{41000000}{16501517}$, which can be typed in as follows: `41000000/16501517`

A correct answer is $\frac{200}{3}$, which can be typed in as follows: `200/3`

A correct answer is $\frac{1000000}{37}$, which can be typed in as follows: `1000000/37`

Question **3**

Correct

Mark 1.00 out of 1.00

Assume you're using a computer write a 2-way set associate 32KB cache where writes that miss in the cache are directly written to the next cache hierarchy (i.e. a write-around policy). Assume that the constant **N** is very large.

Fill in the loops in a way that minimizes cache misses.

```
double A[N][N];
double B[N][N];

void foo()
{
    int i, j;
    [              ] ✓ {
    [for(i = 0; i < N; i ++) {] ✓ {
      [for(j = 0; j < N; j++) {]j];
      }
    }
}
```

Your answer is correct.

The correct answer is:
Assume you're using a computer write a 2-way set associate 32KB cache where writes that miss in the cache are directly written to the next cache hierarchy (i.e. a write-around policy). Assume that the constant **N** is very large.

Fill in the loops in a way that minimizes cache misses.

```
double A[N][N];
double B[N][N];

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

Question **4**

Correct

Mark 1.00 out of 1.00

Given the following definition of structs

```
#define N 1000
typedef struct {
   int vel[3];
   int acc[3];
} point;
points p[N];
```

and the three following routines (which all do the same thing):

(a) clear1

```
void clear1(point *p, int n) {
   int i, j;
   for (i = 0; i < N; i++) {
      for (j = 0; j < 3; j++)
         p[i].vel[j] = 0;
      for (j = 0; j < 3; j++)
         p[i].acc[j] = 0;
    }
}
```

(b) clear2

```
void clear2(point *p, int n) {
   int i, j;
   for (i = 0; i < N; i++) {
      for (j = 0; j < 3; j++) {
         p[i].vel[j] = 0;
         p[i].acc[j] = 0;
    }
}
```

(c) clear3

```
void clear1(point *p, int n) {
   int i, j;
   for (j = 0; j < 3; j++) {
      for (i = 0; i < N; i++)
         p[i].vel[j] = 0;
      for (i = 0; i < 3; i++)
         p[i].acc[j] = 0;
    }
}
```

Best spatial locality	clear1 ✓
Worst spatial locality	clear3 ✓
Not best and not worst locality	clear2 ✓

Your answer is correct.

The correct answer is: Best spatial locality → clear1, Worst spatial locality → clear3, Not best and not worst locality → clear2

Question **5**

Partially correct

Mark 3.00 out of 4.00

Determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b) for a 1024-byte cache using 32-bit memory addresses, 4-byte cache blocks and a single (direct-mapped) set.

This cache has S= 1 ✖ sets, t= 22 ✓ tag bits, s= 8 ✓ set index bits and b= 2 ✓ block offset bits.

Question **6**

Partially correct

Mark 3.64 out of 4.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 10 bits wide.
- . The cache is 2-way associative cache (E=2), with a 8-byte block size (B=8) and 4 sets (S=4).

The following figure shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO - The cache block offset

CI - The cache set index

CT - The cache tag

9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CI	CI	CO	CO	CO
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

A cache with this configuration could store a total of 1024 ✗ bytes of memory (ignoring the tags and valid bits).

Question **7**

Partially correct

Mark 3.60 out of 4.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 9 bits wide.
- . The cache is 2-way associative cache (E=2), with a 4-byte block size (B=4) and 2 sets (S=2).

The following figure shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO - The cache block offset

CI - The cache set index

CT - The cache tag

8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CI	CO	CO
✓	✓	✓	✓	✓	✓	✓	✓	✓

A cache with this configuration could store a total of 512 ✗ bytes of memory (ignoring the tags and valid bits).

Question **8**

Correct

Mark 4.00 out of 4.00

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1024-byte direct-mapped data cache with 16-byte blocks (B = 16).

You are given the following definitions:

```
struct algae_position {
   int x;
   int y;
};

struct algae_position grid[16][16];
int total_x = 0, total_y = 0;
int i, j;
```

When the following code is executed:

```
for (i = 0; i < 16; i++) {
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
  }
}
for (i = 0; i < 16; i++) {
  for (j = 0; j < 16; j++) {
    total_y += grid[i][j].y;
   }
}
```

there are 512 ✓ total reads or loads and 256 ✓ reads or loads that miss in the cache,

resulting in a cache miss rate of 50 ✓ %.

Question **9**

Correct

Mark 4.00 out of 4.00

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640 × 480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines.

The C structures you are using are as follows:

```
struct pixel {
  char r;
  char g;
  char b;
  char a;
};
struct pixel buffer[480][640];
int i,j;
char *cptr;
int *iptr;
```

And assuming the following:

- **sizeof(int) == 4** and **sizeof(char) == 1**
- **buffer** begins at memory address 0
- The cache is initially empty
- The only memory accesses are to the entries of the array **buffer**, with the variables **i, j, cptr and iptr** being stored in registers

Determine the cache performance of the following code:

```
for (i = 0; i < 480; i++) {
   for (j = 0; j < 640; j++) {
     buffer[i][j].r = 0;
     buffer[i][j].g = 0;
     buffer[i][j].b = 0;
     buffer[i][j].a = 0;
  }
}
```

The miss rate is 25 ✓ % (accurate to +/-0.5%)

Each pixel structure is 4 bytes, so each 4-byte cache line holds exactly one structure. For each structure, there is a miss, followed by three hits, for a miss rate of 25%.

Question **10**

Partially correct

Mark 1.00 out of 5.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 12 bits wide.
- . The cache is 4-way associative cache (E=4), with a 16-byte block size (B=16) and 16 sets (S=16).
- The cache contents are as shown below

Set #	Way #0	Way #1	Way #2	Way #3
0:	V=1;Tag=0x4; Data = 0xd7 0xd3 0x7e 0x3b 0x09 0x07 0x84 0x09 0x06 0x37 0xe6 0xfb 0xcb 0xba 0x9a 0x33	V=0;Tag=0xb; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x7; Data = 0xd8 0x9b 0x60 0xe0 0xa8 0x2c 0x89 0x6b 0xe0 0x5f 0xb3 0xf2 0x6f 0x9d 0xb6 0x99	V=1;Tag=0x0; Data = 0x81 0xb8 0x6e 0x4a 0x47 0x96 0xfc 0x94 0x4e 0x04 0xf9 0xe2 0x85 0xb1 0x3e 0x19
1:	V=0;Tag=0x1; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0xe; Data = 0x0a 0xc2 0x22 0x95 0xa2 0x28 0x6c 0x56 0x68 0xfd 0x00 0x2f 0xfd 0xb8 0x4a 0x2a	V=1;Tag=0x6; Data = 0x65 0x51 0xc4 0x1a 0x8e 0x4b 0x9d 0x40 0x3d 0xb0 0x18 0x5f 0xc7 0x32 0xb1 0x36	V=1;Tag=0xb; Data = 0x92 0x57 0x07 0x31 0x78 0xd6 0x39 0x44 0x45 0x03 0xf2 0xaf 0x45 0x6f 0x7b 0xde
2:	V=1;Tag=0x1; Data = 0x5d 0x57 0xa0 0xfa 0x83 0x2c 0x69 0xdc 0xd1 0xea 0xe8 0xdc 0xbb 0x25 0x14 0x1a	V=0;Tag=0xa; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x4; Data = 0x8d 0xa8 0x12 0x06 0xc3 0x66 0xf6 0xc9 0x36 0x20 0xd8 0x89 0xe1 0x5f 0x8a 0x2d	V=1;Tag=0xc; Data = 0x79 0x09 0x15 0xda 0x73 0x50 0xb5 0xcc 0x65 0x0b 0x72 0x28 0x2b 0xbf 0xa5 0xaf
3:	V=1;Tag=0x9; Data = 0x46 0x54 0x0d 0x00 0x86 0x3e 0xce 0x21 0x25 0x16 0xd6 0x00 0x13 0xfc 0xd5 0x57	V=1;Tag=0x7; Data = 0x53 0xd5 0x24 0x65 0x52 0x4d 0x47 0xde 0xe2 0x16 0xa1 0xb9 0x08 0xa4 0x75 0x03	V=1;Tag=0xe; Data = 0xb8 0xa6 0x4f 0xdf 0x08 0xf3 0x1c 0x0a 0x19 0x48 0x34 0x4a 0x3a 0x4c 0x9f 0x89	V=1;Tag=0xb; Data = 0xa1 0x30 0x6c 0xed 0xab 0x2f 0xbc 0xbb 0x4b 0xc3 0x85 0x9b 0x25 0x3d 0xec 0xf4
4:	V=1;Tag=0x0; Data = 0x3c 0xc0 0xb7 0x7f 0x12 0x36 0xe4 0x90 0xc1 0xea 0xe1 0x32 0xc2 0x8a 0x84 0x29	V=1;Tag=0x6; Data = 0x91 0x8b 0xa3 0x05 0x6d 0x1f 0xc2 0xed 0x73 0x23 0xec 0xb6 0x1a 0x9e 0x7e 0x00	V=1;Tag=0xd; Data = 0xf8 0x12 0xda 0x4c 0xa5 0xdd 0xce 0x88 0xf3 0x65 0x26 0xba 0x47 0x5c 0x45 0xcc	V=1;Tag=0xb; Data = 0xfc 0x64 0x8b 0xd6 0xf9 0x21 0x80 0x85 0xf8 0x5b 0x0f 0x22 0x7b 0x05 0x58 0x21
5:	V=1;Tag=0x1; Data = 0x45 0xe3 0x1c 0x4d 0x87 0xcb 0xac 0x37 0x88 0xe3 0x86 0xeb 0xbb 0x10 0xf7 0x45	V=1;Tag=0x9; Data = 0xf2 0xc0 0x5c 0x78 0xb3 0x61 0x28 0x66 0x24 0xad 0x1d 0x15 0x35 0x69 0xa8 0x09	V=1;Tag=0x5; Data = 0x4f 0x8d 0xe1 0x2a 0x47 0x7b 0x49 0x38 0xbf 0x4b 0xc9 0xdc 0xf8 0x09 0xb8 0xb4	V=1;Tag=0x4; Data = 0xcd 0x9d 0x81 0xde 0x1f 0x7b 0x6d 0xaa 0x84 0xe0 0x02 0xa0 0x46 0x72 0xda 0x66
6:	V=1;Tag=0x5; Data = 0xb7 0xe6 0xb2 0xd4 0xc9 0x5f 0x83 0x14 0x9b 0xa6 0x46 0xba 0x23 0x36 0xef 0x9a	V=0;Tag=0x0; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0xb; Data = 0xf6 0xda 0xe8 0xd2 0xe8 0xd6 0x86 0xfd 0x62 0x8a 0xde 0xa6 0xf5 0x8c 0xd9 0x5a	V=1;Tag=0x1; Data = 0x23 0x23 0xc0 0x30 0x47 0x82 0x55 0x79 0x7a 0x90 0xdc 0x87 0xfe 0x01 0x46 0x6d
7:	V=0;Tag=0xd; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x0; Data = 0xbd 0x94 0x3f 0x88 0x99 0xd8 0x89 0xd3 0xdc 0x6d 0x69 0x1f 0xcb 0x71 0x6f 0x9d	V=1;Tag=0xb; Data = 0x1c 0xe4 0xbb 0xec 0x1f 0x09 0x01 0xb1 0x5e 0xb2 0x33 0x04 0xa0 0x02 0x0d 0xad	V=0;Tag=0x8; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
8:	V=1;Tag=0x3; Data = 0x22 0x5c 0xe1 0xcb 0x3c 0x5b 0x48 0x88 0xd1 0x6d 0xe4 0x62 0x6e 0x68 0x9c 0xd1	V=1;Tag=0x6; Data = 0x53 0x60 0x78 0x5a 0xc6 0xf3 0xad 0x1c 0xfe 0xd8 0x83 0x7f 0xf3 0xaa 0xd6 0x2e	V=1;Tag=0x1; Data = 0x5e 0x3c 0x39 0x3c 0x7b 0x43 0xdd 0x71 0xc6 0x7f 0x69 0x1b 0x45 0xb1 0x8d 0x6b	V=1;Tag=0x9; Data = 0x76 0x3e 0xfd 0xd6 0xd4 0x15 0x1c 0xa4 0x61 0x84 0x7c 0x17 0x02 0x4d 0x85 0xaa
9:	V=1;Tag=0x4; Data = 0x0c 0x87 0x5b 0x92 0xc7 0x56 0x87 0x17 0x97 0x0a 0x58 0xc9 0x18 0x0f 0x60 0x49	V=1;Tag=0x7; Data = 0xdd 0xbe 0x78 0x6b 0x57 0xec 0x3e 0x74 0x8c 0x54 0xe2 0x54 0xfe 0x30 0x33 0x25	V=1;Tag=0x6; Data = 0x29 0x18 0x9a 0x81 0x7d 0xd5 0x65 0xaf 0x72 0x82 0x11 0x67 0xb3 0xe7 0xd0 0xa0	V=1;Tag=0x0; Data = 0xeb 0x27 0x7e 0x2a 0x39 0x65 0xe8 0x86 0x62 0x25 0x99 0x2b 0x53 0x0a 0xad 0xbf
10:	V=1;Tag=0x5; Data = 0x34 0xf9 0x01 0x23 0x09 0xdf 0xa4 0xa4 0x07 0xa5 0x3d 0x2b 0xb5 0x6f 0x15 0xfe	V=0;Tag=0xe; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x7; Data = 0x53 0xc0 0xa8 0xe2 0xea 0x94 0xf7 0xb2 0x7a 0x3d 0x9d 0x5c 0x9f 0xbf 0x15 0x4d	V=0;Tag=0x2; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

11:	V=1;Tag=0x3; Data = 0xd4 0xa5 0x00 0x57 0x60 0x47 0x2b 0xde 0x05 0x66 0xce 0xd7 0x95 0x3a 0x8d 0x5d	V=1;Tag=0x2; Data = 0xa8 0x94 0x64 0xbb 0x70 0x36 0x51 0x5c 0xbc 0x36 0xa0 0xa9 0xf9 0xa8 0x37 0xd6	V=1;Tag=0x8; Data = 0xb4 0xc8 0xcc 0x84 0xac 0x48 0xa7 0xb1 0x66 0xf6 0x25 0xa4 0x1d 0x75 0xc9 0xca	V=0;Tag=0xc; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
12:	V=1;Tag=0xc; Data = 0x93 0xe0 0xa7 0xe5 0xf6 0x85 0x0e 0xc3 0x77 0x28 0x6f 0xb7 0x51 0x7e 0xf5 0xba	V=1;Tag=0x1; Data = 0xd9 0x05 0x84 0x45 0x9b 0xba 0xac 0xab 0xc9 0xec 0x99 0xab 0xf7 0x93 0x01 0x20	V=1;Tag=0x2; Data = 0x2f 0xef 0xcc 0xfa 0x4e 0x1a 0x06 0xb6 0xaf 0x8b 0xe9 0x18 0xd9 0x2d 0xae 0xcb	V=1;Tag=0xd; Data = 0x12 0x40 0x6d 0x7b 0x45 0x9d 0x3b 0xfa 0x95 0xd9 0xd0 0xe6 0x1d 0xb2 0x59 0xa9
13:	V=1;Tag=0x4; Data = 0x4a 0xe7 0x90 0x7a 0x13 0x39 0x47 0xf6 0x05 0x39 0x2c 0x43 0x45 0x1b 0x25 0xd6	V=1;Tag=0x3; Data = 0x6a 0x3d 0x71 0x40 0x75 0xbb 0xba 0x1d 0x8d 0xfc 0x94 0x38 0x9a 0xb8 0x32 0x24	V=0;Tag=0x6; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x2; Data = 0xa7 0xc8 0xce 0x0a 0x0a 0x1c 0xa7 0x45 0x07 0x71 0xfd 0x4c 0xe4 0xc3 0x6f 0xeb
14:	V=1;Tag=0x2; Data = 0xe2 0x5a 0x00 0x75 0xa9 0xdf 0x04 0xe5 0x04 0xdf 0xf8 0x3b 0xde 0xcb 0x70 0xb7	V=0;Tag=0x9; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x0; Data = 0xb3 0x63 0x30 0xaa 0x03 0xa0 0xe0 0x2a 0x77 0xc9 0x0b 0xb1 0x38 0xca 0x86 0x18	V=1;Tag=0x3; Data = 0x9b 0x81 0x91 0x8a 0xfd 0x3a 0xe1 0x97 0xa9 0xb8 0xc4 0x82 0x38 0xd1 0xf1 0x55
15:	V=1;Tag=0x3; Data = 0x60 0x6f 0xe1 0x61 0x0c 0x04 0x75 0x48 0xea 0xb7 0x82 0x6c 0x9b 0x75 0xda 0xa4	V=1;Tag=0x0; Data = 0x79 0x41 0xe1 0xc3 0xac 0x1c 0x64 0xb2 0x67 0x53 0x98 0x5f 0xab 0x6d 0x99 0xf3	V=1;Tag=0xa; Data = 0xe9 0xc0 0x83 0x48 0x3c 0xb6 0x3c 0xa7 0x4f 0xfd 0xde 0xce 0x62 0xef 0x1e 0x74	V=1;Tag=0xe; Data = 0x7b 0x35 0xa2 0xe3 0xab 0x8a 0xdc 0x7c 0x56 0x36 0x3f 0xca 0x99 0x6b 0x67 0xef

Assume that memory address **0xed** has been referenced by a load instruction. Indicate the cache entry accessed and the cache byte value returned **in hex** . Indicate whether a cache miss occurs. If there is a cache miss, enter "-" for the "Cache Byte Returned". For values that need a hexidecimal value, do not enter leading zeros even if leading zeros are shown in the value above.

Cache block Offset (CO)	0x 4	✗
Cache set index (CI)	0x 4	✗
Cache tag (CT)	0x 4	✗
Cache hit (Y/N)?	yes	✓
Cache byte returned	0x ab	✗

Question **11**

Partially correct

Mark 1.00 out of 5.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 12 bits wide.
- . The cache is 4-way associative cache (E=4), with a 16-byte block size (B=16) and 8 sets (S=8).
- The cache contents are as shown below

Set #	**Way #0**	**Way #1**	**Way #2**	**Way #3**
0:	V=1;Tag=0x15; Data = 0x91 0xf0 0xd3 0xd6 0xbc 0x8b 0x7e 0xeb 0x37 0x90 0x52 0xeb 0x4d 0xe9 0xc6 0x9f	V=1;Tag=0x0f; Data = 0x7c 0x5c 0x53 0x9b 0x4b 0x3d 0x5e 0xb9 0x9a 0x2e 0x00 0x12 0x00 0x1a 0x72 0xa9	V=1;Tag=0x14; Data = 0x44 0x46 0xc2 0x69 0x36 0x3e 0xf9 0x18 0x55 0x22 0xc0 0xe2 0xe6 0xc3 0x26 0x79	V=1;Tag=0x1e; Data = 0x1d 0xdd 0x03 0x2a 0x34 0xbd 0xc3 0xfa 0x03 0x71 0x2a 0x42 0xc9 0xd9 0x81 0x55
1:	V=1;Tag=0x0a; Data = 0xee 0x73 0xaf 0x8b 0x3d 0xce 0x55 0x7b 0x8a 0x7f 0xbd 0xca 0x4c 0x11 0x5d 0xde	V=1;Tag=0x14; Data = 0x15 0x0e 0x4f 0xf8 0xb9 0xbd 0xe2 0xcb 0x1b 0x51 0x2c 0x91 0x35 0x26 0xac 0xe8	V=1;Tag=0x00; Data = 0xe3 0x74 0xe1 0xc4 0xca 0xd6 0xe2 0x5d 0x52 0xcd 0x62 0xda 0x68 0x44 0xa8 0x2a	V=1;Tag=0x08; Data = 0xdd 0x9b 0x74 0x5f 0x8b 0x13 0x9e 0x9d 0x19 0xcc 0x0e 0x3e 0x68 0x07 0xb0 0xc2
2:	V=1;Tag=0x1d; Data = 0xc0 0xad 0xe1 0x74 0x7f 0xbe 0x73 0xcb 0xae 0xa9 0x07 0x5b 0xfc 0x32 0xb6 0x4c	V=1;Tag=0x08; Data = 0xb1 0xee 0x4b 0x65 0xf6 0x97 0x23 0x32 0x80 0xad 0xa7 0xd6 0x74 0x9e 0xc2 0x07	V=0;Tag=0x0f; Data = -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --	V=1;Tag=0x05; Data = 0x48 0x2e 0x68 0x90 0xca 0x5d 0x0b 0x7f 0x1e 0x2b 0xb3 0xdd 0x4b 0x5e 0xb3 0x9c
3:	V=1;Tag=0x1c; Data = 0xd6 0x6e 0x13 0x82 0x47 0xa6 0x69 0xfb 0xb0 0x23 0x22 0x16 0xce 0xb0 0x30 0x1c	V=1;Tag=0x17; Data = 0xee 0x20 0x42 0x86 0x6e 0x50 0x66 0x50 0x6a 0xba 0x32 0xe3 0xb8 0x6d 0x9a 0x96	V=1;Tag=0x11; Data = 0x8a 0x4b 0x91 0xc2 0x8b 0x4b 0xf9 0xa7 0x19 0x99 0x8c 0x51 0xdd 0xc5 0xbf 0x55	V=1;Tag=0x0d; Data = 0xca 0x2b 0xa1 0x06 0x70 0xd4 0x74 0x81 0xb0 0x0d 0x4e 0x26 0x73 0x73 0x42 0x13
4:	V=1;Tag=0x01; Data = 0xfe 0x04 0x7e 0x4f 0xf1 0x14 0xc0 0xf7 0x2f 0x3e 0xa3 0x46 0x04 0xe3 0xeb 0xc3	V=1;Tag=0x1e; Data = 0x33 0x4e 0x57 0x73 0xe6 0x3f 0x06 0x1f 0xa6 0x81 0x29 0xe8 0xe9 0x90 0xd3 0x0b	V=1;Tag=0x00; Data = 0xf3 0x39 0x7f 0x6a 0xd7 0x96 0xf8 0x42 0x41 0xa2 0x37 0x9a 0xd3 0x9c 0x6d 0x53	V=1;Tag=0x0b; Data = 0xf8 0x3d 0x79 0x92 0x87 0x1e 0x7b 0x45 0x9d 0x6e 0x16 0xab 0x77 0x30 0xb7 0x7a
5:	V=1;Tag=0x1b; Data = 0x10 0x24 0xc7 0x55 0xb6 0x3e 0x77 0xa8 0x92 0xed 0x59 0x04 0xa5 0x56 0xce 0x80	V=1;Tag=0x09; Data = 0x96 0x7b 0xdc 0x9f 0x16 0x2c 0x2f 0xd0 0x12 0x1c 0x98 0x7f 0x2a 0x0b 0x05 0x92	V=1;Tag=0x0d; Data = 0xa9 0xc9 0x7d 0x88 0x92 0x4e 0x8a 0x43 0xc1 0x8d 0xf6 0x34 0x16 0xaa 0x91 0xa1	V=1;Tag=0x14; Data = 0x18 0x73 0x76 0x3e 0x9d 0x96 0x59 0xe4 0xb6 0x25 0x53 0x72 0x54 0x55 0x19 0xc9
6:	V=1;Tag=0x1e; Data = 0x23 0xdf 0xe5 0xeb 0x34 0x7c 0x6d 0x77 0xcb 0xb4 0x06 0x7e 0x21 0xb7 0x66 0xed	V=1;Tag=0x13; Data = 0x30 0x5f 0x0a 0xdb 0xb7 0x4b 0xf0 0xd5 0xb2 0x3d 0x83 0xd7 0xd9 0x6c 0x56 0xa6	V=1;Tag=0x09; Data = 0x10 0x1a 0xda 0xa0 0xbd 0x61 0x2f 0x2d 0x50 0x1d 0x1a 0xe4 0x86 0x7c 0x26 0x0a	V=1;Tag=0x14; Data = 0x96 0x2c 0xf0 0x6f 0xf8 0x17 0xac 0x23 0xe3 0x0e 0xaa 0x39 0x7c 0xca 0x1e 0xed
7:	V=1;Tag=0x18; Data = 0xa6 0x91 0x53 0x9c 0x86 0xb0 0xd6 0x52 0x6f 0xe5 0x41 0xd6 0xe0 0xea 0xdc 0xbb	V=1;Tag=0x08; Data = 0xbc 0x6d 0x7e 0x62 0x6a 0x01 0x74 0x14 0xf3 0x57 0x64 0x80 0x5b 0x3f 0xfd 0xc9	V=1;Tag=0x1c; Data = 0x02 0x7e 0xb9 0xb7 0xaf 0x43 0x45 0x79 0x1b 0xa0 0x4e 0x61 0xeb 0x93 0x2d 0xf2	V=1;Tag=0x0a; Data = 0xf8 0x3a 0x5b 0xcb 0x20 0x22 0x2b 0x13 0x6a 0x1b 0x76 0x31 0x79 0x36 0xd4 0x51

Assume that memory address **0xf64** has been referenced by a load instruction. Indicate the cache entry accessed and the cache byte value returned **in hex** . Indicate whether a cache miss occurs. If there is a cache miss, enter "-" for the "Cache Byte Returned". For values that need a hexidecimal value, do not enter leading zeros even if leading zeros are shown in the value above.

Cache block Offset (CO)	0x 4	✓
Cache set index (CI)	0x 2	✗
Cache tag (CT)	0x 30	✗
Cache hit (Y/N)?	no	✗
Cache byte returned	0x -	✗

Question **12**

Partially correct

Mark 4.00 out of 5.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 10 bits wide.
- . The cache is 2-way associative cache (E=2), with a 4-byte block size (B=4) and 8 sets (S=8).
- The cache contents are as shown below

Set #	**Way #0**	**Way #1**
0:	V=1;Tag=0x14; Data = 0xfa 0x1e 0x87 0x40	V=1;Tag=0x17; Data = 0x22 0x1c 0xa6 0xcd
1:	V=1;Tag=0x00; Data = 0x05 0x26 0x3b 0x01	V=1;Tag=0x11; Data = 0x14 0x1a 0xe1 0x5a
2:	V=1;Tag=0x0d; Data = 0x6f 0x98 0x08 0x9f	V=1;Tag=0x04; Data = 0x1c 0x2f 0xa8 0x2f
3:	V=1;Tag=0x15; Data = 0x37 0x78 0xd0 0x19	V=1;Tag=0x0c; Data = 0xdd 0x74 0x5c 0x4d
4:	V=0;Tag=0x02; Data = -- -- -- --	V=1;Tag=0x09; Data = 0x71 0xb3 0x83 0x4e
5:	V=1;Tag=0x00; Data = 0x62 0xfb 0x1e 0xdb	V=1;Tag=0x05; Data = 0xc2 0x49 0x5e 0x82
6:	V=1;Tag=0x19; Data = 0xed 0x20 0xdb 0x7c	V=1;Tag=0x16; Data = 0x57 0x17 0xc8 0x18
7:	V=1;Tag=0x03; Data = 0x2f 0x81 0xa9 0x07	V=1;Tag=0x14; Data = 0x40 0x92 0x59 0x03

Assume that memory address **0x29** has been referenced by a load instruction. Indicate the cache entry accessed and the cache byte value returned **in hex** . Indicate whether a cache miss occurs. If there is a cache miss, enter "-" for the "Cache Byte Returned". For values that need a hexidecimal value, do not enter leading zeros even if leading zeros are shown in the value above.

Cache block Offset (CO)	0x 1	✓
Cache set index (CI)	0x 2	✓
Cache tag (CT)	0x 1	✓
Cache hit (Y/N)?	no	✓
Cache byte returned	0x 14	✗