Taken from Exam 2, Fall 2003

Problem 3. (12 points):

This problem tests your understanding of basic cache operations. Harry Q. Bovik has written the mother of all game-of-life programs. The Game-of-life is a computer game that was originally described by John H. Conway in the April 1970 issue of Scientific American. The game is played on a 2 dimensional array of cells that can either be alive (= has value 1) or dead (= has value 0). Each cell is surrounded by 8 neighbors. If a life cell is surrounded by 2 or 3 life cells, it survives the next generation, otherwise it dies. If a dead cell is surrounded by exactly 3 neighbors, it will be born in the next generation.

Harry uses a very, very large $N \times N$ array of `int`'s, where N is an integral power of 2. It is so large that you don't need to worry about any boundary conditions. The inner loop uses two int-pointers src and dst that scan the cell array. There are two arrays: src is scanning the current generation while dst is writing the next generation. Thus Harry's inner loop looks like this:

```
...
   int *src, *dst;
...
  {     int n;

    /* Count life neigbors */
    n  = src[ 1    ];
    n += src[ 1 - N];
    n += src[   - N];
    n += src[-1 - N];
    n += src[-1    ];
    n += src[-1 + N];
    n += src[     N];
    n += src[ 1 + N];

    /* update the next generation */
    *dst = (((*src != 0) && (n == 2)) || (n == 3)) ? 1 : 0;

    dst++;
    src++;
  }
...
```

You should assume that the pointers src and dst are kept in registers and that the counter variable n is also in a register. Furthermore, Harry's machine is fairly old and uses a write-through cache with no-write-allocate policy. Therefore, you do *not* need to worry about the write operation for the next generation.

Each cache line on Harry's machine holds 4 int's (16 Bytes). The cache size is 16 KBytes, which is too small to hold even one row of Harry's game of life arrays. Hint: each row has N elements, where N is a power of 2.

Figure 1 shows how Harry's program is scanning the game of life array. The thick vertical bars represent the boundaries of cache lines: four consecutive horizontal squares are one cache line. A neighborhood consists of the 9 squares (cells) that are not marked with an X. The single gray square is the `int` cell that is currently pointed to by src.

The 2 neighborhoods shown in Figure 1 represent 2 successive iterations (case A and B) through the inner loop. The src pointer is incremented one cell at a time and moves from left to right in these pictures.

You shall mark each of the 9 squares those with either a 'H' or a 'M' indicating if the corresponding memory read operation hits (H) or misses (M) in the cache. Cells that contain an X do not belong to the neighborhood that is being evaluated and you should not mark these.

Part 1

In this part, assume that the cache is organized as a direct mapped cache. Please mark the left column in Figure 1 with your answer. The right column may be used as scratch while you reason about your answer. We will grade the left column only.

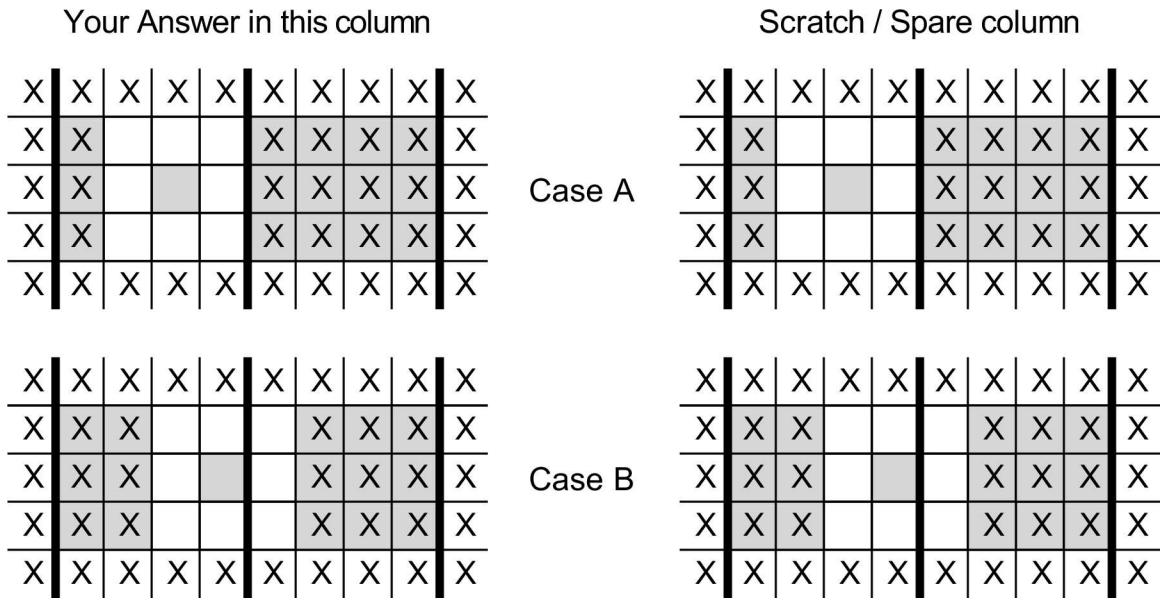


Figure 1: Game of Life with a direct mapped cache

Part 2

In this part, assume a 3-way, set-associative cache with true Least Recently Used replacement policy (LRU). As in Part 1 of this question, please provide your answer by marking the empty squares of the left column in Figure 2 with your solution.

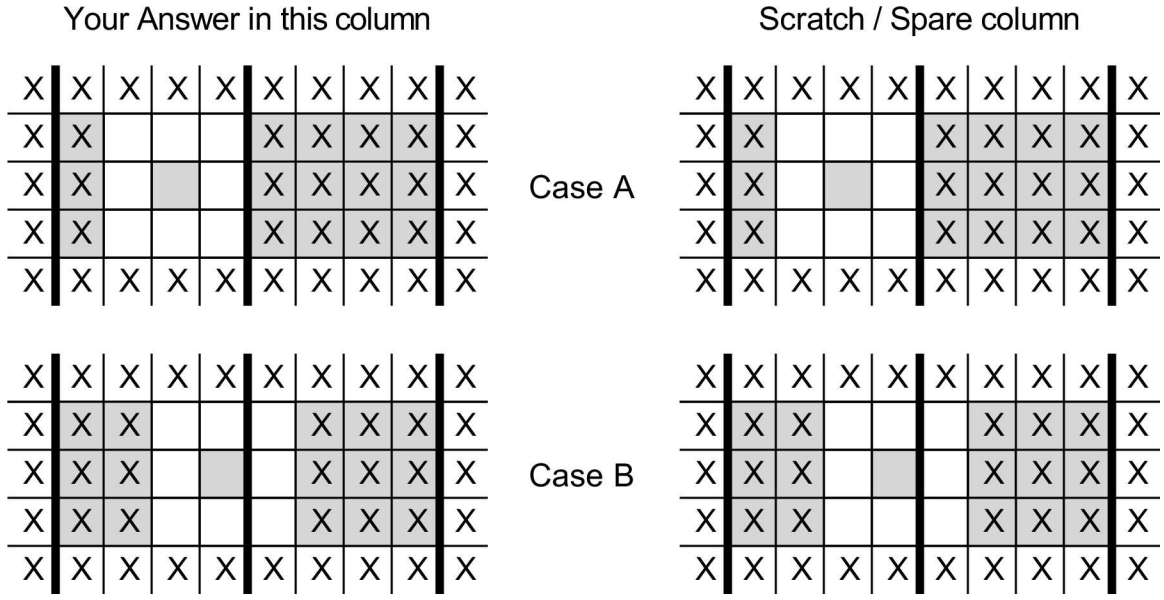


Figure 2: Game of Life with a set associative cache

Taken from Exam 2, Spring 2004

Problem 1. (16 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

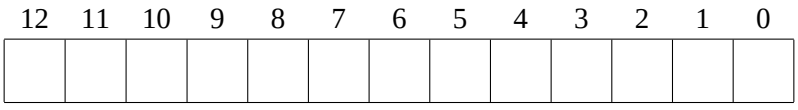
The contents of the cache are as follows:

4-way Set Associative Cache									
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	84	1	ED 32 0A A2	9E	0	BF 80 1D FC	10	0	EF 09 86 2A	E8	0	25 44 6F 1A
1	18	1	03 3E CD 38	E4	0	16 7B ED 5A	02	0	8E 4C DF 18	E4	1	FB B7 12 02
2	84	0	54 9E 1E FA	84	1	DC 81 B2 14	48	0	B6 1F 7B 44	89	1	10 F5 B8 2E
3	92	0	2F 7E 3D A8	9F	0	27 95 A4 74	57	1	07 11 FF D8	93	1	C7 B7 AF C2
4	84	1	32 21 1C 2C	FA	1	22 C2 DC 34	73	0	BA DD 37 D8	28	1	E7 A2 39 BA
5	A7	1	A9 76 2B EE	73	0	BC 91 D5 92	28	1	80 BA 9B F6	6B	0	48 16 81 0A
6	8B	1	5D 4D F7 DA	29	1	69 C2 8C 74	B5	1	A8 CE 7F DA	BF	0	FA 93 EB 48
7	84	1	04 2A 32 6A	96	0	B1 86 56 0E	CC	0	96 30 47 F2	91	1	F8 1D 42 30

Part 1

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

O The block **o**ffset within the cache line
I The cache **i**ndex
T The cache **t**ag

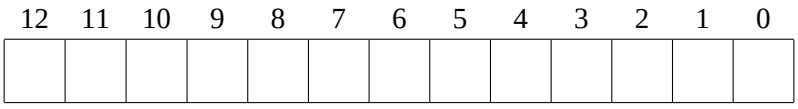


Part 2

For the given physical address, 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 "Cache Byte returned".

Physical address: `0x0D74`

Physical address format (one bit per box)

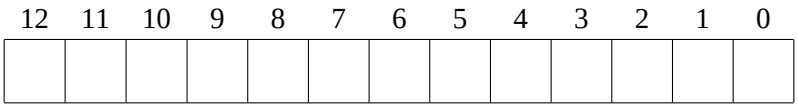


Physical memory reference

Parameter	Value
Cache Offset (CO)	0x_____
Cache Index (CI)	0x_____
Cache Tag (CT)	0x_____
Cache Hit? (Y/N)	_____
Cache Byte returned	0x_____

Physical address: `0x0AEE`

Physical address format (one bit per box)



Physical memory reference

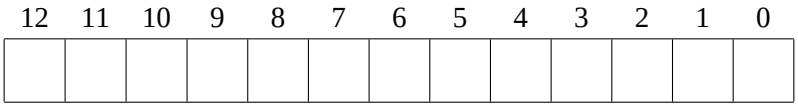
Parameter	Value
Cache Offset (CO)	0x_____
Cache Index (CI)	0x_____
Cache Tag (CT)	0x_____
Cache Hit? (Y/N)	_____
Cache Byte returned	0x_____

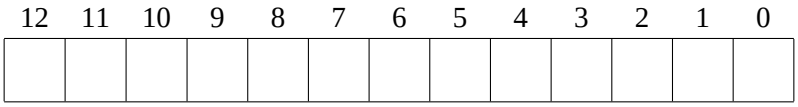
Part 3

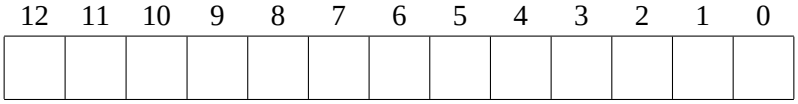
For the given contents of the cache, list all of the hex physical memory addresses that will hit in Set 7. To save space, you should express contiguous addresses as a range. For example, you would write the four addresses `0x1314`, `0x1315`, `0x1316`, `0x1317` as `0x1314--0x1317`.

Answer: _____

The following templates are provided as scratch space:





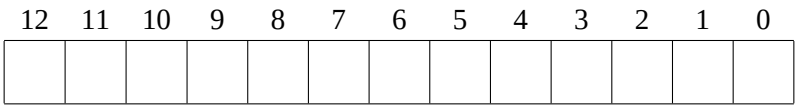


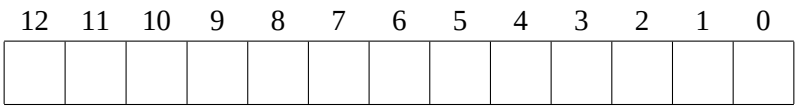
Part 4

For the given contents of the cache, what is the probability (expressed as a percentage) of a cache hit when the physical memory address ranges between `0x1080` - `0x109F`. Assume that all addresses are equally likely to be referenced.

Probability = _____%

The following templates are provided as scratch space:





Game of Life

(1) The first thing to do is figure out the access pattern. There is no way to understand cache performance without first understanding how the memory is accessed. We have a kernel function that goes through the `src` and examines all neighbors.

```
...
int *src, *dst;
...
{
   int n;
   /* Count life neigbors */

   n = src[ 1 ];
   n += src[ 1 - N];
   n += src[ - N];
   n += src[-1 - N];
   n += src[-1 ];
   n += src[-1 + N];
   n += src[ N];
   n += src[ 1 + N];

   /* update the next generation */
   *dst = (((*src != 0) && (n == 2)) || (n == 3)) ? 1 : 0;

   dst++;
   src++;
}
...
```

4	3	2
5	9	1
6	7	8

(2) In the second figure, we are shown how memory is divided into cache lines. These are the two pieces of info that we need: access pattern and memory-cache mappings.

(3) We must use the previous kernel iteration to find out whether we miss on the first access. Since the middle row was the last row accessed, we may assume that it is in the cache at the start of this iteration. We start with a hit.

Direct-Mapped Cache: We can only hold one row from a segment at any given time.

h	h	m
m	m	h
m	h	h

h	m	m
m	m	m
m	h	m

3-Way Set Associative Cache: We can only hold up to 3 rows from a segment at any given time.

h	h	h
h	h	h
h	h	h

h	h	m
h	h	m
h	h	m

Virtual Memory Example

(1) We are first asked to label the different parts of a virtual address. This is not something that we ask you to do for the hell of it. You will need to use this information for the later parts of the question.

(2) The whole idea behind caches is that we come up with imaginary borders in memory. We cut it up into cache *lines*. When we access a memory location, we load the entire line that contains it into the cache. If our programs are written with good spatial locality, then we will get hits on further memory accesses.

(3) Part 1: Each byte in memory has its own address. Given some virtual address, we should be able to narrow down that exact byte that we want. The CO (cache offset) bits are used to index into a specific line once we have loaded it. We know that each cache line is 4 bytes long, so we need 2 bits to index them.

(4) Each line goes into its own set. This figure shows 8 sets, so we need 3 bits to index them. We now know the number of bits taken up by each part of the virtual address: `TTTT TTTT III OO`

Part 2:

- Physical Address: `0x0d74` → `0000 1101 0111 0100`

  `0x0d74` → `0000 1101 0111 0100`→ CT of `0110 1011`, CI of `101`, CO of `00`

  `0x0d74` → `0000 1101 0111 0100`→ CT of `0x6b`, CI of `0x5`, CO of `0x0`

  → valid bit is OFF, the data is dirty.

- Physical Address: `0x0aee` → `0000 1010 1110 1110`

  `0x0aee` → `0000 1010 1110 1110`→ CT of `0101 0111`, CI of `011`, CO of `10`

  `0x0aee` → `0000 1010 1110 1110`→ CT of `0x57`, CI of `0x3`, CO of `0x2`

  → valid bit is ON, data is valid. The byte stored at this memory address is `0xFF`.

(5) Part 3: Set 7 can hold up to 4 valid lines, but here we only see 2. We can use the tag bits to get the address ranges:

One valid tag is (`0x84`) : `0x84` → `1000 0100 111 00 - 1000 0100 111 11` → `0x109c-0x109f`.

One valid tag is (`0x91`) : `0x91` → `1001 0001 111 00 - 1001 0001 111 11` → `0x123c-0x123f`.

(6) Part 4: `0x1080-0x109F` → `1 0000 1000 0000 - 1 0000 1001 1111`: set indices go from 0 to 7. offsets go from 0 to 3. Tag stays fixed as `0x1000 0100` → `0x84`. We can look for this tag and see how many sets contain it with valid data.

We see the tag `0x84` in 4 out of our 8 sets: $4/8 = 50$ percent