Andrew login ID:_______________________________________

Full Name:_______________________________________

CS 15-213, Fall 2003

Exam 2

November 18, 2003

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 66 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

Problem	Score
1 (10):	
2 (10):	
3 (12):	
4 (11):	
5 (09):	
6 (08):	
7 (06):	
TOTAL (66):	

Problem 1. (10 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	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	0C	0	03 3E CD 38	A0	0	16 7B ED 5A	8A	1	8E 4C DF 18	E4	1	FB B7 12 02
2	8A	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	1	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	8A	1	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BE	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	8A	1	48 16 81 0A
6	38	1	5D 4D F7 DA	82	1	69 C2 8C 74	8A	1	A8 CE 7F DA	3E	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	06	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:

CO The block offset within the cache line
CI The cache index
CT The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0

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: `0x1314`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

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: `0x08DF`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

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 3. 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:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

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 `0x1140` - `0x115F`. Assume that all addresses are equally likely to be referenced.

Probability = ___________%

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 2. (10 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 22 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 2048 bytes.
- The TLB is 2-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	003	EB	1
	007	46	0
1	028	D3	1
	001	2F	0
2	031	E0	1
	012	D3	0
3	001	5C	0
	00B	D1	1
4	02A	BA	0
	011	F1	0
5	01F	18	1
	002	4A	1
6	007	63	1
	03F	AF	0
7	010	0D	0
	032	10	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	37	1	10	16	0
01	58	1	11	37	0
02	19	1	12	28	0
03	2A	1	13	53	0
04	56	0	14	1D	0
05	33	0	15	4A	1
06	61	0	16	49	0
07	28	0	17	26	0
08	42	0	18	0C	1
09	63	0	19	04	1
0A	31	1	1A	1F	0
0B	5C	0	1B	22	1
0C	5A	1	1C	40	0
0D	2D	0	1D	0E	1
0E	4E	0	1E	35	1
0F	1D	1	1F	03	1

A. Part 1

(a) The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) 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:

PPO The physical page offset
PPN The physical page number

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. Part 2

For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: `0x005EEC`

(a) Virtual address format (one bit per box)

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(c) Physical address format (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Virtual address: `0x00AF9D`

(a) Virtual address format (one bit per box)

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(c) Physical address format (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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.

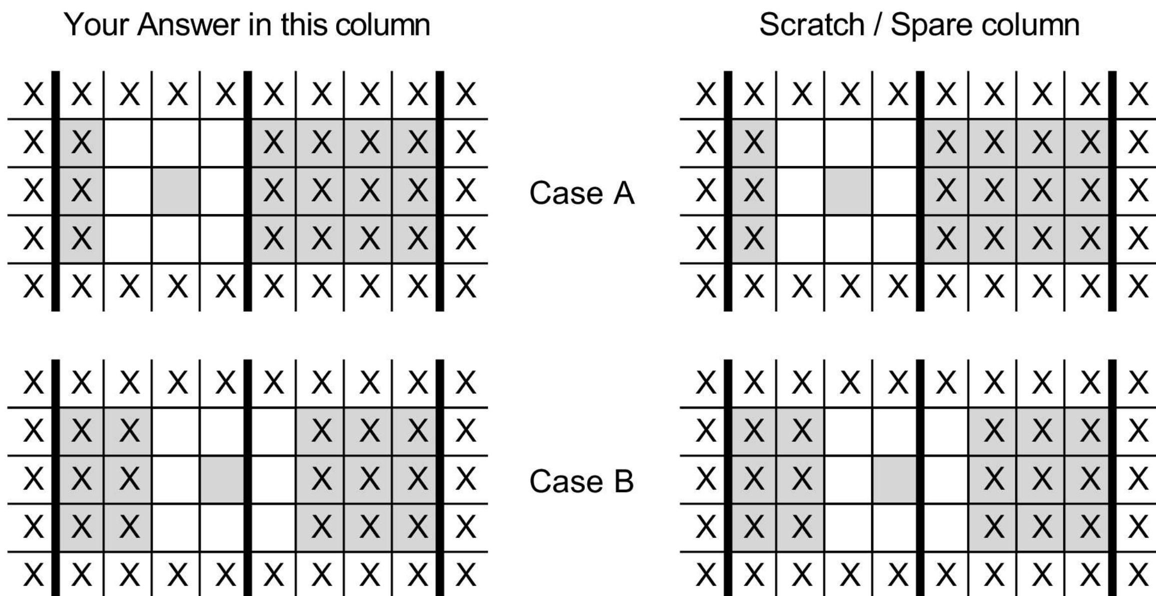

Figure 1: Game of Life with a direct mapped cache

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.

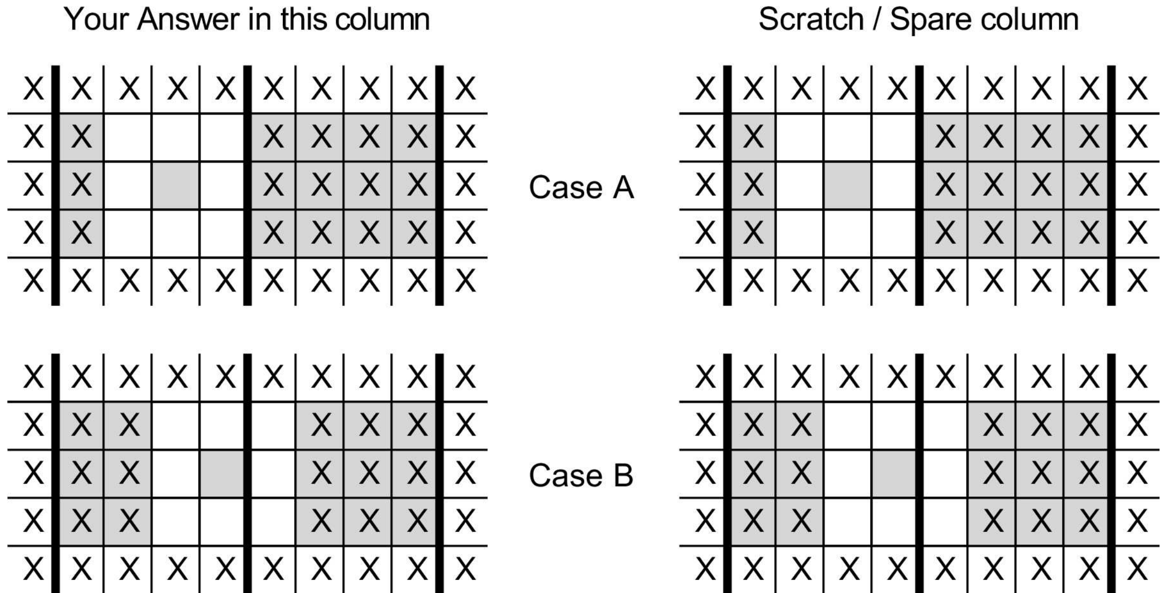

Figure 2: Game of Life with a set associative cache

Problem 4. (11 points):

This problem requires you to analyze the behavior of the inner loops from a simple linear algebra package. The int-vector X of length N is added to all rows of the $N \times N$ int-matrix A:

```
 1 ...
 2    int X[N] = {0};
 3    int A[N][N] = {0};
 4 ...
 5   {   int i, j;
 6 ...
 7       for (i = 0; i < N; i++) {
 8         for (j = 0; j < N; j++) {
 9           int t;
10           t = A[i][j];
11           t += X[j];
12           A[i][j] = t;
13         }
14      }
15 ...
16   }
```

X is allocated first and is directly followed by the matrix A. In other words, the address of $X[N]$ is the address of $A[0][0]$. You may assume that X is aligned so that $X[0]$ maps to the first set of the cache.

Part 1

Assume a 1K byte direct-mapped cache with 16 byte blocks. Each `int` is 4 bytes long. You should assume that i, j, and t are kept in registers and are not polluting the cache. Furthermore, you should ignore all instruction fetches: you are only concerned with the data cache.

Fill in the table below for the following problems size, estimating the miss-rate expressed as a percentage of all load/store operations to X and A. For the percentage, 2 digits of precision suffices.

N	Total # of memory refs to A and X	# of misses to X	# of misses to A	Miss rate (in %)
8				
64				
60				

Part 2

Consider lines 10-12 of the program. How can you rewrite this part of the loop to improve performance?

Part 3

A Victim-Cache (VC) is a small extra cache that is used for lines that are evicted from the main cache. VC's tend to be small and fully associative. On a read, the processor is looking up the victim cache and the main cache in parallel. If there is a hit in the victim cache, the line is transferred back to the main cache and the line from the main cache is now stored in the VC (line swap). On a cache miss (both VC and main cache), the data is fetched from memory and placed in the main cache. The evicted line is then copied to the VC. Assume that this machine has a 1-line victim cache. What is the miss rate of this system for $N = 64$?

Miss rate = ___________________%

Problem 5. (9 points):

This problem tests your understanding of memory bugs. Each of the code sequences below may or may not contain memory bugs. The code all compiles without warnings or errors. If you think there is a bug, please circle **YES** and indicate the type of bug from the list below of memory bugs. Otherwise, if you think there are no memory bugs in the code, please circle **NO**.

Bugs:

1. Potential buffer overflow error
2. Memory leak
3. Potential for dereferencing a bad pointer
4. Incorrect use of free
5. Incorrect use of realloc
6. Misaligned access to memory
7. Other memory bug

Part A

```
/*
 * strndup - An attempt to write a safe version of strdup
 *
 * Note: For this problem, assume that if the function returns a
 * non-NULL pointer to dest, then the caller eventually frees the dest buffer.
 */
char *strndup(char *src, int max)
{
    char *dest;
    int i;

    if (!src || max <= 0)
        return NULL;
    dest = malloc(max+1);
    for (i=0; i < max && src[i] != 0; i++)
        dest[i] = src[i];
    dest[i] = 0;
    return dest;
}
```

NO **YES** Type of bug: ____________________

Part B

```
/* Note: For this problem, asssume that if the function returns a non-NULL
 * pointer to node, then the caller eventually frees node. */
struct Node {
    int data;
    struct Node *next;
};

struct List {
    struct Node *head;
};

struct Node *push(struct List *list, int data)
{
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));

    if (!(list && node))
        return NULL;
    node->data = data;
    node->next = list->head;
    list->head = node;
    return node;
}
```

NO **YES** Type of bug: ____________________

Part C

```
/* print_shortest - prints the shortest of two strings */
void print_shortest(char *str1, char *str2)
{
    printf("The shortest string is %s\n", shortest(str1, str2));
}

char *shortest(char *str1, char *str2)
{
    char *equal = "equal";
    int len1 = strlen(str1);
    int len2 = strlen(str2);

    if (len1 == len2)
        return equal;
    else
        return (len1 < len2 ? str1 : str2);
}
```

NO **YES** Type of bug: ____________________

Problem 6. (8 points):

This problem tests your understanding of Unix process control. Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main()
{
  int status;
  int counter = 2;
  pid_t pid;

  if ((pid = fork()) == 0) {
      counter += !fork();
      printf("%d", counter);
      fflush(stdout);
      counter++;
  }
  else {
      if (waitpid(pid, &status, 0) > 0) {
          printf("6");
          fflush(stdout);
      }
      counter += 2;
  }

  printf("%d", counter);
  fflush(stdout);
  exit(0);
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

```
A. 264343      Y    N
B. 236434      Y    N
C. 243643      Y    N
```

How many possible strings start with 234... ? (Give only the number of strings.)

Answer = _______

Problem 7. (6 points):

Suppose the file `foo.txt` contains the text "`yweixtr`", `bar.txt` contains the text "`ounazvs`", and `baz.txt` does not yet exist. Examine the following C code, and answer the two questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {
    int fd1, fd2, fd3, fd4, status;
    pid_t pid;
    char c;

    fd1 = open("foo.txt", O_RDONLY, 0);
    fd2 = open("foo.txt", O_RDONLY, 0);
    fd3 = open("bar.txt", O_RDONLY, 0);
    fd4 = open("baz.txt", O_WRONLY | O_CREAT, DEF_MODE); /* r/w */

    dup2(fd4, STDOUT_FILENO);

    if ((pid = fork()) == 0) {
        dup2(fd3, fd2);
        dup2(STDOUT_FILENO, fd4);
        read(fd1, &c, 1);
        printf("%c", c);
        read(fd2, &c, 1);
        printf("%c", c);
        read(fd3, &c, 1);
        printf("%c", c);
        printf("\n");
        exit(0);
    }

    waitpid(pid, &status, 0);
    read(fd1, &c, 1);
    printf("%c", c);
    read(fd2, &c, 1);
    printf("%c", c);
    read(fd3, &c, 1);
    printf("%c", c);
    printf("\n");
    return 0;
}
```

A. What will the contents of `baz.txt` be after the program completes?

B. What will be printed on `stdout`?

```
A. Contents of baz.txt:                  B. Printed on stdout:
```