Andrew login ID:_____

Full Name:_____

CS 15-213, Fall 2003

Final Exam

December 9, 2003

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- The exam has a maxi
- This exam is OPEN B ou may use a calculator, but no laptops or other wi

1 (08):
2 (09):
3 (04):
4 (08):
5 (08):
6 (10):
7 (08):
8 (09):
9 (06):
10 (08):
11 (10):
TOTAL (88):

Problem 1. (8 points):

Consider the following m-bit floating-point representation based on the IEEE floating-point format:

- There is a sign bit-field in the most significant bit *s*.
- The next k bit-fields are the exponent *exp*.
- The last n bit-fields are the significand *frac*.

In this format, a given numeric value V is encoded in the form $V = (-1)^s \times M \times 2^E$, where s is the sign bit, E is exponent after biasing, and M is the significand.

Part I

Give a formula for the largest odd integer that can be represented exactly.

Part II

Give a formula for the smallest positive normalized value.

The following data structure declarations pertain to the next problem.

(Feel free to remove this page from your exam packet for easy reference.)

```
struct s1 {
  char a[3];
  union u1 *b;
  int c;
};

struct s2 {
  struct s1 d;
  struct s1 *e;
  struct s2 *f;
  double g;
  int h[4];
};

union u1 {
  int i;
  struct s2 j;
  struct s1 *k;
};
```

Problem 2. (9 points):

In the following problem, you are given the task of reconstructing C code based on the declarations of C structures and unions from the previous page, and the IA32/Linux assembly code generated when compiling the C code.

For each IA32 assembly code sequence below on the left, fill in the missing portion of corresponding C source line on the right.

A proc1:

```
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 4(%eax),%eax
    movl 40(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
int proc1(struct s1 *x)
{
  return x->____________________ ;
}
```


B proc2:

```
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 32(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```



```
                                        _____ ;
}
```


C proc3:

```
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 4(%eax),%eax
    movl (%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
int proc3(union u1 *x)
{
  return x->____________________ ;
}
```

Problem 3. (4 points):

This problem concerns the indexing of C arrays.

Consider the C code below, where `N` is a constant declared with `#define`.

```
int foo (int A[16][N], int i, int j)
{
    return A[i][j];
}
```

Suppose the above C code generates the following assembly code:

```
foo:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%ecx
  movl 16(%ebp),%edx
  movl 12(%ebp),%eax
  sall $2,%edx
  sall $3,%eax
  subl 12(%ebp),%ea
  leal (%edx,%eax,4
  movl %ebp,%esp
  popl %ebp
  movl (%ecx,%eax),%eax
  ret
```


What is the value of `N`?

`N` =

Problem 4. (8 points):

Consider the following four C and IA32 functions. Next to each of the four IA32 functions, write the name of the C function that it implemnts. If the assembly routine doesn't match any of the above functions, write NONE. To save space, the startup code for each IA32 function is omitted:

```
   pushl %ebp
   movl %esp,%ebp
```

```
int *winter(int foo[12][8],
            int i, int j)
{
  return &foo[i][j];
}
```

```
movl    8(%ebp), %eax
movl    12(%ebp), %edx
movl    (%eax,%edx,4), %edx
movl    16(%ebp), %eax
popl    %ebp
leal    (%edx,%eax,4), %eax
ret
```



```
int *spring(int foo[1
            int i, int j)
{
  return foo[i+j];
}
```



```
movl    16(%ebp), %eax

sall    $5, %e
addl    %edx
ret
```



```
int *summer(int** foo,
            int i, int j)
{
  return &foo[i][j];
}
```

```
movl    12(%ebp), %edx
movl    16(%ebp), %eax
addl    %edx, %eax
movl    8(%ebp), %edx
popl    %ebp
movl    (%edx,%eax,4), %eax
ret
```

```
int *fall(int** foo,
          int i, int j)
{
  return foo[i+j];
}
```

```
movl    12(%ebp), %eax
movl    16(%ebp), %ecx
movl    8(%ebp), %edx
sall    $3, %eax
popl    %ebp
addl    %ecx, %eax
sall    $2, %eax
addl    %edx, %eax
ret
```

Problem 5. (8 points):

This problem tests your understanding of basic cache operations.

(Note: This is the same problem from Exam 2, with one exception. In Exam 2 we asked you to complete two iterations, say k and $k+1$, of the game. In this problem, we are asking you to do the next two iterations, $k+2$ and $k+3$.

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 neighbors */
    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 center 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

For the next problem, you are given four different cache organizations. All four caches are of the same size, namely 1024 bytes. However the caches are organized differently:

A. **Cache A:** is a direct mapped cache with a line size of 8 bytes (= 2 **`int's`**).

B. **Cache B:** is a 4-way, set-associative cache with a line size of 8 bytes (= 2 **`int's`**) and least recently used (LRU) replacement policy.

C. **Cache C:** is a direct mapped cache with a line size of 64 bytes (= 16 **`int's`**).

D. **Cache D:** is a 4-way, set-associative cache with a line size of 64 bytes (= 16 **`int's`**) and LRU replacement policy.

(Feel free to remove this page from your exam packet for easy reference.)

Problem 6. (10 points):

This problem tests your understanding of how the cache organization can impact the performance of a program.

For each kernel listed below, you should determine which of the 4 cache organizations from the previous page performs best, circling the letters (A, B, C, D) associated those cache organizations.

For this problem we define *"best"* to mean that the cache has the least number of cache misses. You should assume that the caches are cold prior to executing the kernels. In other words, the caches have no valid data and the first access to any datum will cause a cache miss. In some cases, "best" is not unique, so that there are two or more cache organizations that perform equally well. In this case, you must list all cache organizations that have the same performance for full credit.

For example, a kernel that touches only one variable will always miss on that access, no matter how the cache is organized. So the correct answer would be to circle A, B, C, and D.

Each kernel has some loop variables (**`i,j`**) and a working variable (**`x`**), which are kept in registers and which do not cause any memory accesses. Likewise, you are supposed to ignore instruction fetches.

1. **Kernel 1 (2 pts)**:

```
int A[127][127];
...
{
  int i, j, x = 0;

  for (i = 0; i < 127; i++)
    for (j = 0; j < 127; j++)
      x += A[j][i];

  return x;
}
```

   The best cache organization(s) is(are): A B C D

2. **Kernel 2 (2 pts)**:

```
int A[127][127];
...
{
  int i, j, x = 0;

  for (i = 0; i < 127; i++)
    for (j = 0; j < 127; j++)
      x += A[i][j];

  return x;
}
```

   The best cache organization(s) is(are): A B C D

3. **Kernel 3 (3 pts)**:

```
int A[127][127], B[127][127];
...
{
  int i, j, x = 0;

  for (i = 0; i < 127; i++)
    for (j = 0; j < 127; j++)
      x += A[i][j] * B[i][j];

  return x;
}
```

The best cache organization(s) is(are): A B C D

4. **Kernel 4 (3 pts)**:


```
int A[16][16], B[16][16];
...
{
  int i, j, x = 0;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)
      x += A[j][i] * B[i][j];

  return x;
}
```

The best cache organization(s) is(are): A B C D

The following C program and declarations are part of the next problem. For each part, the three comment lines

```
/* LINE 1 */
/* LINE 2 */
/* LINE 3 */
```

will be replaced with different fragments of code. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int counter = 2;

void foo() {
    counter++;
    printf("%d", counter);
}

int main() {
    pthread_t tid[2
    int i;

    for (i = 0; i < 2; i++) {

        /* LINE 1 */
        /* LINE 2 */
        /* LINE 3 */

    }

    counter++;
    printf("%d", counter);
}
```


(Feel free to remove this page from your exam packet for easy reference.)

Problem 7. (8 points):

This problem tests your understanding of the differences between processes and threads.

Part 1

Suppose the following code replaces the three comment lines in the program on the previous page:

```
LINE 1:
LINE 2:    Pthread_create(&tid[i], 0, foo, 0);
LINE 3:
```

What is the **first** number that gets printed on stdout? Circle only one answer.

3
4
5
Could be either 3 or 4 or 5
Could be either 3 or 4

Part 2

Suppose the following code replaces the three comment lines:

```
LINE 1:    Pthread_create(&tid[i], 0, foo, 0);
LINE 2:    Pthread_join(tid[i], 0);
LINE 3:
```

What is the **first** number that gets printed on stdout? Circle only one answer.

3
4
5
Could be either 3 or 4 or 5
Could be either 3 or 4

Part 3

Suppose the following code replaces the three comment lines.

```
LINE 1:    if (fork() == 0) {
LINE 2:        foo();
LINE 3:    }
```

What is the **first** number that gets printed on stdout? Circle only one answer.

```
3
4
5
Could be either 3 or 4 or 5
Could be either 3 or 4
```


Part 4

Consider the **same** cod n stdout? Circle only one answer.


```
3
4
5
Could be either 3 or 4 or 5
Could be either 3 or 4
```

Problem 8. (9 points):

This problem tests your understanding of signals.

For each of the code segments below, circle the **largest** value that could be printed to stdout. Remember that when the system executes a signal handler, it blocks signals of the type currently being handled (and no others).

```
int i = 0;

void handler(int s){
  if( !i ){
    kill(getpid(),
         SIGINT);
  }
  i++;
}

int main(){
  signal(SIGINT,
         handler
  kill(getpid(),
       SIGINT);

  printf("%d\n",i);

  return 0;
}
```

- 0
- 1
- 2
- 3
- 4
- 5
- 1000

```
int i = 0;

void handler(int s){
  if( !i ){
    kill(getpid(),
         SIGINT);
    kill(getpid(),
         SIGINT);
  }
  i++;
}

  kill(getpid(),
       SIGINT);
  printf("%d\n",

  return 0;
}
```

- 0
- 1
- 2
- 3
- 4
- 5
- 1000

```
int i = 0;

void handler(int s){
  if( !i ){
    kill(getpid(),
         SIGINT);
    kill(getpid(),
         SIGUSR1);
  }
  i++;
}

     n(){
   nal(SIGINT,
       handler);
   nal(SIGUSR1,
       handler);
       tpid(),
       GUSR1);

  printf("%d\n",i);

  return 0;
}
```

- 0
- 1
- 2
- 3
- 4
- 5
- 1000

Problem 9. (6 points):

This problem tests your understanding of pointer arithmetic and pointer dereferencing.

Harry Q. Bovik has decided to exercise his creativity and has created the most exotic dynamic memory allocator that the 213 staff has ever seen. The following is a description of Harry's block structure:

HDR	ID_STRING	PAYLOAD	FTR

- HDR - Header of the block (4 bytes)
- ID_STRING - Unique ID string (8 bytes)
- PAYLOAD - Payload of the block (arbitrary size)
- FTR - Footer of the block (4 bytes)

The size of the payload of each block is stored in the header and the footer of the block. Since there is an 8 byte alignment requirement, the least significant of the 3 unused bits is used to indicate whether the block is free (0) or allocated (1). Harry ing stored right after the header of the block. The siz

For this problem, you can ass

- sizeof(int) == 4 bytes
- sizeof(char) == 1 bytes
- sizeof(short) == 2 bytes
- The size of any pointer (e.g. char*) is 4 bytes.

Your task is to help Harry figure out and circle **clearly** which of the following definitions of the macro `GET_ID` will cause `print_block()` to output the string that is stored in the `ID_STRING` field. **There may be multiple macros that are correct, so be sure to circle all of them.**

Also, assume that the block pointer `bp` points to the first byte of the payload.

```
/* Harry Q. Bovik's print_block() function
   Refer to this function in order to figure out
   the context in which the GET_ID macro is used.
*/
void print_block(void *bp){
  printf("Found block ID: %s\n", GET_ID(bp));
}

/* A. */
#define GET_ID(bp)  ((char *)(((int)bp) - 8))

/* B. */
#define GET_ID(bp)  ((char *)(((char)bp) - 8))

/* C. */
#define GET_ID(bp)  ((c

/* D. */
#define GET_ID(bp)  ((char *)((char *)bp) - 8))

/* E. */
#define GET_ID(bp)  ((char *)(((int *)bp) - 4))

/* F. */
#define GET_ID(bp)  ((char *)(((int *)bp) - 8))

/* G. */
#define GET_ID(bp)  ((char *)(((char**)bp) - 8))

/* H. */
#define GET_ID(bp)  ((char *)(((short*)bp) - 4))

/* I. */
#define GET_ID(bp)  ((char *)(((short*)bp) - 8))
```

Problem 10. (8 points):

Suppose the file `foo.txt` contains letters, and, `bar.txt` contains numbers. Examine the following C code, and answer the questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {

    int fda, fdb, fdc, pid;
    char c;

    fda = open("foo.txt", O_RDONLY, 0);
    fdb = open("foo.txt", O_RDONLY, 0);
    fdc = open("bar.txt", O_RDONLY, 0);

    if ((pid = fork()) == 0) {
        dup2(fdb, fdc);
        read(fda, &c, 1);
        read(fdb, &c, 1);
        read(fdc, &
    }

    if (pid)
        wait(0);
    dup2(fda, fdb);
    read(fda, &c, 1);
    read(fdb, &c, 1);
    read(fdc, &c, 1);
    close(fdb);
    fdb = open("bar.txt", O_RDONLY, 0);
    read(fda, &c, 1);
    read(fdb, &c, 1);
    read(fdc, &c, 1);

    exit(0);
}
```



```
Immediately before the child exits:
  How many letters have been read so far?                    _________

  How many numbers have been read so far?                    _________

Immediately before the parent exits:
  How many letters have been read since the child exited?    _________

  How many numbers have been read since the child exited?    _________
```

Problem 11. (10 points):

This problem tests your understanding of concurrency and synchronization.

Below are some code segments that use threads. For each segment, list all possible output number sequences that could be printed. If a code segment possibly does not output any numbers, write "NONE" as one of the possibilities.

Note: You may assume that the code contains no errors other than the ones that may arise due to concurrency issues. Also assume that all thread library calls always work without any errors. Lastly, assume no optimization is done during compilation.

Code Segment 1

```
void *square(void *x)
{
  int *y = x;
  printf("%d", *y * (*y));
  return NULL;
}

int main()
{
  pthread_t t;
  int i = 2;

  pthread_create(&t, NULL, square, &i);
  printf("%d", ++i);

  pthread_join(t, NULL);
  return 0;
}
```


Possible output sequences:

Code Segment 2

```
/* The following code is a simple simulation of a bar:
 *   - each thread represents a customer
 *   - visit() represents a customer's visiting the bar
 *   - occupancy represents current number of customers in the bar
 *   - bartender represents the person in charge of the bar
 */
sem_t bartender;
int occupancy=0;

void *visit(void *customerID)
{
  sem_wait(&bartender);   /* P(&bartender) */
  occupancy++;

  if ((int)customer == -1)
    printf("%d", occupancy);

  occupancy--;
  sem_post(&barte

  return NULL;
}

int main()
{
  pthread_t t[5];
  int i;

  sem_init(&bartender, 0, 2); /* initialized with the value 2 */

  /* let customers in */
  for (i=0; i < 5; i++)
    pthread_create(&t[i], NULL, visit, (void *)i);

  visit((void *)-1);

  return 0; /* close down the bar */
}
```


Possible output sequences: