Andrew login ID:_______________________________________

Full Name:_______________________________________

CS 15-213, Fall 2004

Final Exam

December 16, 2004

Instructions:

- Make sure that your exam has 24 pages and 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 123 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 not use a calculator, laptop or any other electronic or wireless device. Good luck!

Problem	Score
1 (10):	
2 (12):	
3 (14):	
4 (6):	
5 (8):	
6 (8):	
7 (8):	
8 (9):	
9 (9):	
10 (13):	
11 (10):	
12 (6):	
13 (10):	
TOTAL (123):	

Problem 1. (10 points):

Consider the following 16-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next $k = 7$ bits are the exponent. The exponent bias is 63.
- The last $n = 8$ bits encode the significand.

Numeric values are encoded in this format as a value of 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

Answer the following problems using either decimal (e.g., 1.375) or fractional (e.g., $11/8$) representations for numbers that are not integers.

For denormalized numbers:

A. What is the smallest value E of the exponent after biasing? ____________

B. What is the largest value M of the significand? ____________

For normalized numbers:

A. What is the smallest value E of the exponent after biasing? ____________

B. What is the largest value E of the exponent after biasing? ____________

C. What is the smallest value M of the significand? ____________

D. What is the largest value M of the significand? ____________

Part II

Suppose we want to implement this representation in C. One way to do this would be with the following struct. The 16-bit float is stored as two unsigned chars (assumed to be 8 bits each).

```
typedef struct
{
   unsigned char exps;
   unsigned char frac;
} float16 ;
```

You are now asked to finish implementing a function that takes a `float16` **and divides it by 2**. The value returned by `div2` should follow the IEEE format restrictions that we described earlier for this 16-bit number. **You may assume that the argument `f` is non-negative**.

```
float16  div2(float16  f)
{
   /* save copies so we can modify struct fields */
   unsigned char frac = f.frac;
   unsigned char exps = f.exps;

   /* check for infinity and NaN */
   if ( exps == ____________ )
   {
      /* in that case, division gives us same number */
      f.frac = frac;
      f.exps = exps;
   }

   /* check for denormalized numbers */
   if ( exps == ____________ )
   {
      f.frac = ____________ ;
      f.exps = ____________ ;
   }

   /* check for a normalized number that becomes denormalized */
   if ( exps == ____________ )
   {
      f.frac = ____________ ;
      f.exps = ____________ ;
   }

   if ( ____________ && ____________ ) /* remaining cases */
   {
      f.frac = ____________ ;
      f.exps = ____________ ;
   }

   return f;
}
```

Problem 2. (12 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system with the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 4096 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	16	13	1
	1B	2D	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0
4	06	08	1
	0F	19	1
5	0A	09	1
	1F	20	1
6	02	13	0
	18	12	1
7	0C	0B	0
	1E	24	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	2D	0
05	13	1	15	1B	0
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	1
0C	01	1	1C	1E	0
0D	15	1	1D	3E	1
0E	0C	0	1E	27	1
0F	14	0	1F	18	1

Part 1

1. The diagram below shows the bits of a virtual address. Please indicate the locations of the following fields by placing an 'X' in the corresponding boxes of that field's row. For example, if the virtual page offset were computed from the 2 most significant bits of the virtual address, you would mark the 'O' (offset) column as shown:

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

O The virtual page **o**ffset
N The virtual page **n**umber
I The TLB **i**ndex
T The TLB **t**ag

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
O																				
N																				
I																				
T																				

2. The diagram below shows the format of a physical address. Please indicate the locations of the following fields by placing an 'X' in the corresponding boxes of that field's row.

O The physical page **o**ffset
N The physical page **n**umber

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

Part 2

For the given virtual addresses, please 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 the physical address blank.

Virtual address: `0x00123`

1. Virtual address (one bit per box)

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

2. Address translation

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

3. Physical address(one bit per box)

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

Virtual address: `0x02321`

1. Virtual address (one bit per box)

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

2. Address translation

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

3. Physical address(one bit per box)

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

Problem 3. (14 points):

Short-answer questions, two points each. Write at most one line's worth of text to each question. Make sure that your answer is legible: use the scratch space at the top until you're sure of your answer. We will not grade your scratch space.

A. Briefly explain why a segregated list allocator is almost always faster than an equally well-written explicit freelist allocator.

B. Name the most important thing a process has that a thread lacks.

C. Name one advantage of increasing the page size for virtual memory. Name one disadvantage.

D. Name one advantage of dynamic linking over static linking. Name one disadvantage.

E. Normally, the exponent field of a floating point number is interpreted as $E - bias$; but in denormalized numbers (when $E = 0$), it is interpreted as $0 - bias + 1$. Explain why.

F. In network code using sockets, what condition would cause your process to receive a SIGPIPE?

G. I'm writing a code to run a large simulation. I have made changes which I hope will halve the running time, and I try to get timings on the program by running it on my laptop. After one hour I suspend it (which sends SIGSTOP to all programs) and go home, where some hours later I unsuspend it (sending SIGCONT); the program terminates a few minutes later. Name one timing measure which should still indicate a large speedup, and another which will be useless.

Problem 4. (6 points):

This problem tests your understanding of signals, setjmp() and longjmp().

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <signal.h>

struct list_s {
    int data;
    struct list_s *next;
};

static jmp_buf jmpEnv;

static void handleSEGV(int signal) {
    longjmp(jmpEnv, 1);
}

struct list_s *pushlist(struct list_s *head, int value) {
    struct list_s *ele;

    ele = (struct list_s *)malloc(sizeof(struct list_s));
    if(ele == NULL) {
        return head;
    }

    ele->next = head;

    ele->data = value;
    return ele;
}

int func(struct list_s *head) {
    volatile int sum = 0; /* 'sum' is not kept in a register */

    setjmp(jmpEnv);
    if(sum) return sum;

    while(1) {
        sum += head->data;
        head = head->next;
    }
}
```

```
int main() {
    struct list_s *list = NULL;

    signal(SIGSEGV, handleSEGV);

    list = pushlist(list, 2);
    list = pushlist(list, 4);
    list = pushlist(list, 6);

    printf("%d\n", func(list));
    return 0;
}
```

A. What is printed by this program?

B. Explain briefly what happens if you call `func(NULL)`.

Problem 5. (8 points):

A. You are given the following loop:

```
for(prod = 1, i = 0; i < 128; i++) {
   prod = prod * vec[i];
}
```

That compiles into the following assembly code:

```
  movl $0x1, %eax
  movl $0x0, %esi
.L2:
  imul 0xfffffde8(%ebp,%esi,4),%eax
  incl %esi
  cmpl $0x80, %esi
  jl   .L2
```

Calculate the CPE (cycles per element) of this loop assuming 4 cycles for a multiply instruction, 3 cycles for a memory load, and 1 cycle for all other instructions. The processor is fully pipelined, and can issue multiple instructions every cycle. For simplicity, assume there is an unbounded number of functional units. Also assume that the processor makes no branch mispredictions.

B. Assume the loop is then unrolled as follows:

```
for(prodA = 1, prodB = 1, prodC = 1, prodD = 1, i = 0; i < 126; i += 3) {
   prodA = prodA * vec[i];
   prodB = prodB * vec[i+1];
   prodC = prodC * vec[i+2];
}
prod = vec[i] * vec[i+1];
prod *= prodA * prodB * prodC;
```

That compiles into the following assembly code:

```
  movl $0x1, %eax
  movl $0x1, %ebx
  movl $0x1, %ecx
  movl $0x0, %esi
.L2:
  imul 0xffffde8(%ebp,%esi,4),%eax
  imul 0xffffdec(%ebp,%esi,4),%ebx
  imul 0xffffdf0(%ebp,%esi,4),%ecx
  addl 0x3, %esi

  cmpl $0x7D, %esi
  jl   .L2
  movl 0xffffde8(%ebp,%esi,4),%edx
  imul 0xffffdec(%ebp,%esi,4),%edx
  imul %ebx, %eax
  imul %ecx, %edx
  imul %edx, %eax
```

Calculate the CPE of this unrolled loop using the same assumptions from part 1.

Problem 6. (8 points):

This problem deals with the bit representation of integer values. Assume that all values are 32 bits in length and stored in two's complement form.

Part A:

A. Using only constants and the `<<` and `+` operators, calculate `10 * x` using fewer than 4 operators.

B. Using only constants and the `~` and `+` operators, calculate `a - b`

Part B:

You are given the following code:

```
int func(int x) {
    int a = x | (x >> 16) | (x << 16);
    int b = a | (a >> 8) | (a << 8);
    int c = b | (b >> 4) | (b << 4);
    int d = c | (c >> 2) | (c << 2);
    int e = d | (d >> 1) | (d << 1);

    return e;
}
```

A. What is the output of `func(12)`?

B. What is the output of `func(0)`?

C. Using only constants, the `func()` function, and the `~` and `&` operators, provide code for: `x = (a == 0) ? 1 : 0`.

Problem 7. (8 points):

Time to hand-optimize some code.

A.

```
1  #define N 15213
2  struct vector {
3      int x[N];
4  };

6  int dot(const struct vector *a, const struct vector *b) {
7      int sum = 0;
8      int i;
9      for(i = 0; i < N; i++) {
10         sum += a->x[i] * b->x[i];
11     }
12     return sum;
13 }
```

This code is computing the inner product of two vectors. Unroll the loop (doing two operations per iteration).

```
int dot(const struct vector *a, const struct vector *b) {









}
```

B.

```
1  struct vector {
2      int n;
3      int *x;
4  };

6  int dot(const struct vector *a, const struct vector *b) {
7      int sum = 0;
8      int i;
9      assert(a->n == b->n);
10     for(i = 0; i < a->n; i++) {
11         sum += a->x[i] * b->x[i];
12     }
13     return sum;
14 }
```

This code again computes the inner product of the vector arguments, but this time the vectors have variable length. Find three loop-invariant computations and lift them out of the loop.

```
int dot(const struct vector *a, const struct vector *b) {




}
```

C. I lifted the three loop invariant computations out of the loop, then typed `gcc -O2 dot.c -o dot` but I didn't notice any speedup. Why not?

Problem 8. (9 points):

This problem will test your understanding of x86 assembly code. Consider the following C code:

```
struct list {
  struct list *next;
  int data;
};

struct list *
get_ith_elem(struct list *head, int i) { /*assume i < length of list */
  int j = 0;
  struct list *node = head;
  while(i != j) {
    node = node->next;
    j++;
  }
  return node;
}
```

A. Using the following skeleton, fill in the assembly code that realizes the C function above. Do not use C variable names.

```
1   get_ith_elem:
2          push %ebp
3          movl %esp,%ebp
4          movl 0xc(%ebp),%ecx
5          movl 0x8(%ebp),%eax
6          movl $0,%edx
7          cmpl %edx,%ecx

8          je ________
9     .L1:

10         incl ________

11         cmpl %edx,________

12         movl (________),%eax
13         jne .L1
14    .L2:
15         leave
16         ret
```

B. For each C variable, write the register with which it corresponds in the above assembly code.

```
i       _________

j       _________

node    _________
```

C. Suppose instead we wanted to retrieve the ith node's data field. That is, the final line of the C code would become `return node->data`. In assembly, this can be accomplished by adding one instruction. What is that instruction and between what lines would it be placed?

Problem 9. (9 points):

This question will test your understanding of concurrent programming, specifically deadlocks. For these questions, assume each function is executed by a unique thread on a uniprocessor system.

A. Consider the following C code:

```
1   void thread1() {               12  void thread2() {
2     P(lock1);                    13    P(lock1);
3     P(lock2);                    14    P(lock2);
4     P(lock3);                    15    P(lock3);
5                                  16
6     /* do some work */           17    /* do some work */
7                                  18
8     V(lock2);                    19    V(lock1);
9     V(lock3);                    20    V(lock2);
10    V(lock1);                    21    V(lock3);
11  }                              22  }
```

Does this code contain a deadlock? If so, write a sequence of line numbers that, when executed in that order, will cause the deadlock.

B. Consider the following C code:

```
1   void thread1() {               12  void thread2() {
2     P(lock1);                    13    P(lock3);
3     P(lock2);                    14    P(lock2);
4     P(lock3);                    15    P(lock1);
5                                  16
6     /* do some work */           17    /* do some work */
7                                  18
8     V(lock3);                    19    V(lock3);
9     V(lock2);                    20    V(lock2);
10    V(lock1);                    21    V(lock1);
11  }                              22  }
```

Does this code contain a deadlock? If so, write a sequence of line numbers that, when executed in that order, will cause the deadlock.

C. Consider the following C code:

```
1  void thread1() {                10  void thread2() {
2    P(lock1);                     11    P(lock3);
3    P(lock2);                     12    P(lock1);
4                                  13
5    /* do some work */            14    /* do some work */
6                                  15
7    V(lock2);                     16    V(lock1);
8    V(lock1);                     17    V(lock3);
9  }                               18  }
```

```
19 void thread3() {
20   P(lock2);
21   P(lock3);
22
23   /* do some work */
24
25   V(lock3);
26   V(lock2);
27 }
```

Does this code contain a deadlock? If so, write a sequence of line numbers that, when executed in that order, will cause the deadlock.

Problem 10. (13 points):

The code shown below implements a simple client/server communication protocol. In this protocol, the server sends a block of data using the following format:

```
Content-Length: <n>
<data>
```

where <n> is the number of bytes, and <data> is the set of bytes to be sent. The first line is terminated by a newline ('\n') character.

Here is an initial implementation of the function used by the server to send one block of data:

```
1 /* Send block of data to client */
2 void server_write(int clientfd, char *data, int len)
3 {
4   char buf[MAXBUF];
5   sprintf(buf, "Content-Length: %d\n%s", len, data);
6   Rio_writen(clientfd, buf, strlen(buf));
7 }
```

Here is an initial implementation of the function used by the client to receive all the blocks of data from the server:

```
1 /* Read data sent by server */
2 void client_read(int serverfd)
3 {
4   rio_t rio;
5   char buf[MAXBUF];
6   int n, len;
7   Rio_readinitb(&rio, serverfd);
8   while (Rio_readlineb(&rio, buf, MAXBUF) > 0) {
9     if (sscanf(buf, "Content-Length: %d", &len) > 0) {
10      n = Rio_readn(serverfd, buf, len);
11      buf[n] = '\0'; /* Terminate string */
12      printf("Client received data: '%s'\n", buf);
13    } else {
14      printf("Couldn't determine content length\n");
15    }
16  }
17 }
```

As a test, the following function runs a session in which the server sends as data the strings “`Hello`” and “`World!`”:

```
1 void server_session(int clientfd)
2 {
3   char *words[2] = {"Hello", "World!"};
4   int i;
5   for (i = 0; i < 2; i++) {
6     server_write(clientfd, words[i], strlen(words[i]));
7     sleep(1); /* Ensures that writes don't get combined */
8   }
9 }
```

The call to `sleep` on line 7 of this function simply ensures that each call to `server_write` will generate a separate packet transmission from the server to the client. Otherwise, the operating system tries to coalesce the results of multiple writes into a single packet.

When we run the server and client, the client prints the following result:

```
Client received data: 'Conte'
Couldn't determine content length
Couldn't determine content length
```

A. Explain briefly why the session didn't work the way it should. Include in your explanation why the first block of data received by the client was `'Conte'` rather than `'Hello'`.

B. Show how you could modify the client code to reliably process messages sent according to the protocol. Describe your modification of function `client_read`, referring to the line numbers of the existing code. Give a brief explanation of how this change will make the session work properly.

C. Suppose you have already distributed the client code to millions of customers. You can't repair a buggy client, but you can modify the server code to compensate for the client's shortcomings. Describe your modification of function `server_write`, referring to the line numbers of the existing code. Give a brief explanation of how this change will make the session work properly.

D. What serious security flaw exists in the client function? Describe briefly how it could be exploited by a malicious server.

Problem 11. (10 points):

Suppose we have a simple in-order pipeline processor that blocks on data misses. The data cache of this processor has the following configurations:

Cache size	C bytes
Cache line	B bytes
Cache associativity	4-way, uses LRU replacement policy
Cache latency	H cycles
Latency between cache and memory	P cycles

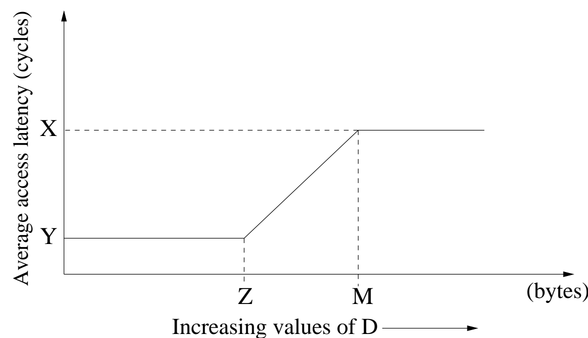
Table 1: 'Cache latency" is the time to fetch data from the cache to the CPU registers; 'Latency between cache and memory" is the time to fetch a cache block from the memory into the cache.

The following C code sequentially accesses D elements of array A for 100 times. To simplify your reasoning, assume the only memory accesses are to the entries of A; A starts at memory address 0; the cache is initially empty before the outer loop; `sizeof(char)=1`.

```
char A[N]; /* N >> D */

int sum = 0;
for (i = 0; i < 100; i ++) {
    for (j = 0; j < D; j ++) {
        sum += A[j];
    }
}
```

We run the above code with various values of D and get the following graph. The horizontal axis is the total number of bytes in the D elements and the vertical axis is the average latency to access an element in the array A.



Determine the following values with the configuration parameters in Table 1 (C, B, 4, H, and P)

A. Y = ___________________ Z = ___________________

B. Number of cache lines = ______________________

C. Number of cache sets = ______________________

D. M = ___________________

E. What's the **hit rate** when D > M (in the range of the second plateau)? hit rate = ___________

F. X = ___________________

Problem 12. (6 points):

This question tests your understanding of file sharing on UNIX systems. Suppose “letter.txt” contains “abcdefg...xyz” and “number.txt” contains “0123...789”. Consider the following code. (We don’t check return values for space reasons. Assume all functions return normally.) Write down the values of “ch” in the comments (an example answer is given at line 13) and answer a short question.

```
1  int main(int argc, char *argv[])
2  {
3    int fd1, fd2, fd3, fd4;
4    char ch;
5
6    fd1 = open("number.txt", O_RDONLY);
7    fd2 = open("number.txt", O_RDONLY);
8
9    fd3 = open("letter.txt", O_RDONLY);
10   fd4 = dup(fd3);
11
12   read(fd1, &ch, 1);
13   printf("%c", ch);      /* ch = '0'  */
14
15   read(fd2, &ch, 1);
16   printf("%c", ch);      /* ch = ____ */
17
18   read(fd3, &ch, 1);
19   printf("%c", ch);      /* ch = ____ */
20
21   read(fd4, &ch, 1);
22   printf("%c", ch);      /* ch = ____ */
23   fflush(stdout);
24
25   if ( fork() == 0 ){
26     read(fd1, &ch, 1);
27     printf("%c", ch);    /* ch = ____ */
28
29     exit(0);
30   }
31   wait(NULL);
32
33   read(fd1, &ch, 1);
34   printf("%c", ch);      /* ch = ____ */
35   fflush(stdout);
36
37   return 0;
38 }
```

Short question: What is the semantic difference between calling open() twice on the same file as opposed to calling open() once and then calling dup()?

Problem 13. (10 points):

This question tests your understanding on how the linker operates. The programs `main.c` and `a.c`, shown in the left- and right-hand columns below, will be compiled separately and then linked into one executable. No other modules or libraries will be used. For each of the variable declarations in the `a.c` file, you need to complete the comment with a note that indicates what will happen:

- **OK-G** : this is a valid declaration of a variable that is global to both `main.c` and `a.c`. Global in the context means that changes made by function `main()` (defined in `main.c`) will be correctly seen code in function `a_func()` (defined in `a.c`), and vice versa.
- **OK-L** : this is a valid declaration of a variable that is local to `a_func()` only. Any changes made by code in `a_func()` will have no side-effects visible to code in `main()`
- **LE** : This declaration will cause a linker error. While each file will compile fine, the attempt to link both object files into one executable will fail and the linker with complain about this symbol.
- **SE** : This declaration will cause a silent error. Neither the compiler nor the linker will notice any problem, but the program will have an obvious bug.
- **OE** : Some other error not covered by the previous 4 cases occurs.

Please clearly insert the 2 or 3 letter abbreviation of one of the 5 cases in each comment field for the program file `a.c` :

main.c:

```
extern int foo;

static double abc;

int x; int *x_ptr = &x;

struct {
    double xyzzy;
} bar;

int cnt = 0;

double eps = 1e-6;

#define MAX_N 99

int dead;

unsigned beaf;

int main() {
    ...
}
```

a.c:

```
double foo;          /*    */

double abc;          /*    */

int *x_ptr;          /*    */

double xyzzy;        /*    */

int bar = 0;         /*    */

int cnt;             /*    */

double eps = 1e-6;   /*    */

int MAX_N = 99;      /*    */

unsigned *dead;      /*    */

static int beaf;     /*    */

int a_func() {
    ...
}
```