Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2006

Final Exam

Thursday Dec 14, 2006

- 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 92 points.
- This exam is OPEN BOOK. You may use any books or notes you like. You may use a calculator, but no other electronic devices. Good luck!

01 (06):	
02 (09):	
03 (06):	
04 (12):	
05 (12):	
06 (06):	
07 (15):	
08 (06):	
09 (04):	
10 (08):	
11 (08):	
TOTAL (92):	

Problem 1. (6 points):

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Numeric values are encoded as a value of the form $V = M \times 2^E$, where E is exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). Any rounding of the significand is based on *round-to-even*.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. Give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$).

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
$1/32$		
1/16		
3/32		
1		
11		
12		

Problem 2. (9 points):

Structs and arrays. The next two problems require understanding how C code accessing structures and arrays is compiled. Assume the x86-64 conventions for data sizes and alignments.

You are given the following C code:

```
#include "decls.h"

typedef struct {
    int x[CNT2];          /* Unknown constant */
    int y;
    int z[CNT3];          /* Unknown constant */
} struct_a;

typedef struct{
    struct_a data[CNT1];  /* Unknown constant */
    int idx;
} struct_b;

void set_y(struct_b *bp, int val)
{
    int idx = bp->idx;
    bp->data[idx].y = val;
}
```

You do not have a copy of the file `decls.h`, in which constants `CNT1`, `CNT2`, and `CNT3` are defined, but you have the following x86-64 code for the function `set_y`:

```
set_y:
    bp in %rdi, val in %esi
  movslq  168(%rdi),%rax
  leaq    (%rax,%rax,2), %rax
  movl    %esi, 12(%rdi,%rax,8)
  ret
```

Based on this code, determine the values of the three constants

A. `CNT1` =

B. `CNT2` =

C. `CNT3` =

Problem 3. (6 points):

Structs and arrays. As in the previous problem, assume the x86-64 conventions for data sizes and alignments.

You are given the following C code:

```
#include "decls.h"

typedef struct{
    type_t x;         /* Unknown type */
    int y[3];
} struct_a;

typedef struct{
    int low;
    struct_a val[N];  /* Unknown constant */
    int high;
} struct_b;

int get_high(struct_b *bp)
{
    return bp->high;
}
```

You do not have a copy of the file `decls.h`, in which constant `N` and data type `type_t` are declared, but you have the following x86-64 code for the function `get_high`:

```
get_high:
    bp in %rdi
  movl    104(%rdi), %eax
  ret
```

Provide *some* valid combination of these two parameters for which the assembly code would be generated.

A. `type_t`:

B. `N` =

Problem 4. (12 points):

Loops. Consider the following x86-64 assembly function, called `looped`:

```
looped:
        # a in %rdi, n in %esi
        movl    $0, %edx
        testl   %esi, %esi
        jle     .L4
        movl    $0, %ecx
.L5:
        movslq  %ecx,%rax
        movl    (%rdi,%rax,4), %eax
        cmpl    %eax, %edx
        cmovl   %eax, %edx
        incl    %ecx
        cmpl    %edx, %esi
        jg      .L5
.L4:
        movl    %edx, %eax
        ret
```

Fill in the blanks of the corresponding C code.

- You may only use the C variable names `n`, `a`, `i` and `x`, not register names.
- Use array notation in showing accesses or updates to elements of `a`.

```
int looped(int a[], int n)
{
    int i;
    int x = ________________;

    for(i = ______________; __________________; i++) {

        if (______________________)

            x = ___________________;
    }
    return x;
}
```

Problem 5. (12 points):

Stack discipline. Below is a segment of code you will remember from your buffer lab, the section that reads a string from standard input.

```
int getbuf() {
    char buf[8];
    Gets(buf);
    return 1;
}
```

The function `Gets` is similar to the library function `gets`. It reads a string from standard input (terminated by `\n` or end-of-file) and stores it (along with a null terminator) at the specified destination. `Gets` has no way of determining whether `buf` is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

Below is the object dump of the `getbuf` function:

```
08048c4b <getbuf>:
 8048c4b: 55                     push   %ebp
 8048c4c: 89 e5                  mov    %esp,%ebp
 8048c4e: 83 ec 38               sub    $0x20,%esp
 8048c51: 8d 45 d8               lea    0xfffffff0(%ebp),%eax
 8048c54: 89 04 24               mov    %eax,(%esp)
 8048c57: e8 f2 00 00 00         call   8048d4e <Gets>
 8048c5c: b8 01 00 00 00         mov    $0x1,%eax
 8048c61: c9                     leave
 8048c62: c3                     ret
```

(over)

Suppose that we set a breakpoint in function `getbuf` and then use gdb to run the program with an input file redirected to standard input. The program stops at the breakpoint when it has completed the `sub` instruction at `0x08048c4e` and is poised to execute the `lea` instruction at `0x08048c51`. At this point we run the following gdb command that lists the 12 4-byte words on the stack starting at the address in `%esp`:

```
0x08048c51 in getbuf ()
(gdb) x/12w $esp
0x55683a58:   0x003164f8    0x00000001    0x55683a98    0x0030bab6
0x55683a68:   0x003166a4    0x555832e8    0x00000001    0x00000001
0x55683a78:   0x55683ab0    0x08048bf9    0x55683ab0    0x0035b690
```

A. What is the address of `buf`? `0x`____________________

B. When the program reaches the breakpoint, what is the value of `%ebp`? `0x`_____________________

C. To which address will `getbuf` return after executing? `0x`_____________________

D. When the program reaches the breakpoint, what is the value of `%esp`? `0x`_____________________

E. Instead of having `getbuf` return to its calling function, suppose we want it to return to a function `smoke` that has the address `0x8048b20`.

Below is an incomplete sequence of the hex values of each byte in the file that was input to the program (we have given you the first 8 padding values). Fill in the remaining blank hex values so that the call to `Gets` will return to `smoke`. Note that `smoke` does not depend on the value stored in `%ebp`.

```
0x01   0x02   0x03   0x04   0x05   0x06   0x07   0x08

0x___  0x___  0x___  0x___  0x___  0x___  0x___  0x___

0x___  0x___  0x___  0x___  0x___  0x___  0x___  0x___
```

Problem 6. (6 points):

Consider the following function for computing the dot product of two arrays of n integers each. We have unrolled the loop by a factor of 3.

```
int dotprod(int a[], int b[], int n)
{
    int i, x1, y1, x2, y2, x3, y3;
    int r = 0;
    for (i = 0; i < n-2; i += 3) {
        x1 = a[i]; x2 = a[i+1]; x3 = a[i+2];
        y1 = b[i]; y2 = b[i+1]; y3 = b[i+2];

        r = r + x1 * y1 + x2 * y2 + x3 * y3;   // Core computation
    }
    for (; i < n; i++)
        r += a[i] * b[i];
    return r;
}
```

Compute the performance of this function in terms of cycles per element (CPE) for each of the following associations for the core computation. Assume that we run this code on a machine in which multiplication requires 7 cycles, while addition requires 5. Further, assume that these latencies are the only factors constraining the performance of the program. Don't worry about the cost of memory references or integer operations, resource limitations, etc.

Re-association	CPE
`((r + x1 * y1) + x2 * y2) + x3 * y3`	
`(r + (x1 * y1 + x2 * y2)) + x3 * y3`	
`r + ((x1 * y1 + x2 * y2) + x3 * y3)`	
`r + (x1 * y1 + (x2 * y2 + x3 * y3))`	
`(r + x1 * y1) + (x2 * y2 + x3 * y3)`	

Problem 7. (15 points):

Cache memories. This problem requires you to analyze both high-level and low-level aspects of caches. You will be required to perform part of a a cache translation, determine individual hits and misses, and analyze overall cache performance.

For this problem, you should assume the following:

- Memory is byte addressable.
- Physical addesses are 14 bits wide.
- The cache is 2-way set associative with an 8 byte block-size and 2 sets.
- Least-Recently-Used (LRU) replacement policy is used.
- `sizeof(int)` = 4 bytes.

(*over)*

A. The following question deals with a matrix declared as `int arr[4][3]`. Assume that the array has already been initialized.

(a) (1 point) 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 set index
`CT` The cache tag

13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) (1 point) Given that the address of `arr[0][0]` has value **0x2CCC**, perform a cache address translation to determine the block offset and set index for the first item in the array.

`CI = 0x________`
`CO = 0x________`

13	12	11	10	9	8	7	6	5	4	3	2	1	0

(c) (3 points) For each element in the matrix `int arr[4][3]`, label the diagram below with the set index that it will map to.

`arr[4][3]`	Col 0	Col 1	Col 2
Row 0			
Row 1			
Row 2			
Row 3			

B. (6 points) The following questions also deals with `int arr[4][3]` and the cache defined at the beginning of the problem. Assume the cache stores only the matrix elements; variables `i`, `j`, and `sum` are stored in registers.

```
int i, j;
int sum = 0;

for(i=0; i<4; i++){
  for(j=0; j<3; j++){
    sum += arr[i][j];
  }
}

/* second access begins */
for(i=2; i>=0; i=i-2){
  for(j=0; j<3; j++){
    sum += arr[i][j];
    sum += arr[i+1][j];
  }
}
/* second access ends */
```

Assume the above piece of code is executed. Fill out the table to indicate if the corresponding memory access will be a hit (**h**) or a miss (**m**) when accessing the array `arr[4][3]` for the **second** time (between the comments 'second access begins' and 'second access ends').

`arr[4][3]`	Col 0	Col 1	Col 2
Row 0			
Row 1			
Row 2	**h**		
Row 3			

The following grids can be used as scrap space:

C. The following question deals with a different matrix, declared as `int arr[5][5]`. Again assume that `i`, `j`, and `sum` are all stored in registers.

Consider the following piece of code:

```
#define ITERATIONS 1
int i, j, k;
int sum = 0;

for(k=0; k<ITERATIONS; k++){
  for(i=0; i<5; i++){
    for(j=0; j<5; j++){
      sum += arr[i][j];
    }
  }
}
```

For each of the following caches, specify the total number of **cache misses** for the above code. **Important:** Assume that the matrix is aligned so that `arr[0][0]` is the first element in a cache block.

(a) (2 points) If ITERATIONS is 1 (Total accesses: 25).

i. Direct-mapped, 16 byte block-size, 4 sets

`Number of cache misses` ________

ii. 2-way set associative, 8 byte block-size, 2 sets

`Number of cache misses` ________

(b) (2 points) If ITERATIONS is 2 (Total accesses: 50).

i. Direct-mapped, 64 byte block-size, 2 sets

`Number of cache misses` ________

ii. 2-way set associative, 32 byte block-size, 1 set

`Number of cache misses` ________

Problem 8. (6 points):

Process control.

A. What are the possible output sequences from the following program:

```
int main() {
    if (fork() == 0) {
        printf("a");
        exit(0);
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
```

Circle the possible output sequences: `abc` `acb` `bac` `bca` `cab` `cba`

B. What is the output of the following program?

```
pid_t pid;
int counter = 2;

void handler1(int sig) {
    counter = counter - 1;
    printf("%d", counter);
    fflush(stdout);
    exit(0);
}

int main() {
    signal(SIGUSR1, handler1);

    printf("%d", counter);
    fflush(stdout);

    if ((pid = fork()) == 0) {
        while(1) {};
    }
    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}
```

OUTPUT: ____________

Problem 9. (4 points):

File I/O. This problem tests your understanding of how Linux represents and shares files. You are asked to show what each of the following programs prints as output:

- Assume that file `infile.txt` contains the ASCII text characters “`15213`”;
- You may assume that system calls do not fail;
- When a process with no children invokes `waitpid(-1,NULL,0)`, this call returns immediately;
- *Hint*: each of the following questions has a unique answer.

A.

```
int main() {
  int fd;
  char c;

  fd = open("infile.txt", O_RDONLY, 0);

  fork();
  waitpid(-1, NULL, 0);

  read(fd, &c, sizeof(c));
  printf("%c", c);

  return 0;
}
```

OUTPUT: ________________

B.

```
int main() {
  int fd;
  char c;

  fork();
  waitpid(-1, NULL, 0);

  fd = open("infile.txt", O_RDONLY, 0);

  read(fd, &c, sizeof(c));
  printf("%c", c);

  return 0;
}
```

OUTPUT: ________________

Problem 10. (8 points):

Concurrency and sharing. Consider a concurrent C program with two threads and a shared global variable `cnt`. The threads execute the following lines of code:

Thread 1	Thread 2
`/* Increment cnt */` `cnt++;`	`/* Decrement cnt */` `cnt--;`

Suppose that these lines of C code compile to the following assembly language instructions:

Thread 1

```
movl cnt,%eax   # L1: Load cnt
inc  %eax       # U1: Update cnt
movl %eax,cnt   # S1: Store cnt
```

Thread 2

```
movl cnt,%eax   # L2: Load cnt
dec  %eax       # U2: Update cnt
movl %eax,cnt   # S2: Store cnt
```

At runtime, the operating system kernel will choose some ordering of these instructions. Since we are not explicitly synchonizing the threads, some of these orderings will produce the correct value for `cnt` and others will not.

Each of the sequences shown below gives a possible ordering of the instructions when the two threads execute. Assuming that `cnt` is initially zero, what is the value of `cnt` in memory after each of the sequences completes?

A. cnt=0; $L_1, U_1, S_1, L_2, U_2, S_2$ `cnt ==` __________

B. cnt=0; $L_1, U_1, L_2, S_1, U_2, S_2$ `cnt ==` __________

C. cnt=0; $L_2, U_2, S_2, L_1, U_1, S_1$ `cnt ==` __________

D. cnt=0; $L_1, L_2, U_2, S_2, U_1, S_1$ `cnt ==` __________

Problem 11. (8 points):

Synchronization. This question will test your understanding of synchronizations, deadlocks and use of semaphores. For these questions, assume each function is executed by a unique thread on a uniprocessor system.

A. Consider the following C code:

```
/* Initialize semaphores */
mutex1 = 1;
mutex2 = 1;
mutex3 = 1;
mutex4 = 1;
```

```
void thread1() {

    P(mutex4); _____
    P(mutex2); _____
    P(mutex3); _____

    /* Access Data */

    V(mutex4); _____
    V(mutex2); _____
    V(mutex3); _____

}
```

```
void thread2() {

    P(mutex1); _____
    P(mutex2); _____
    P(mutex4); _____

    /* Access Data */

    V(mutex1); _____
    V(mutex2); _____
    V(mutex4); _____

}
```

A. Can this code deadlock? Yes No

B. If yes, then indicate a feasible sequence of calls to the P or V operations that will result in a deadlock. Place an ascending sequence number (1, 2, 3, and so on) next to each operation in the order that it is **called**, even if it never returns. For example, if a P operation is called but blocks and never returns, you should assign it a sequence number.

Note that there are several correct solutions to this problem.

B. Consider the following three threads and three semaphores:

```
/* Initialize semaphores */
s1 = 1;
s2 = 0;
s3 = 0;

/* Initialize x */
x = 0;
```

```
void thread1()          void thread2()          void thread3()
{                       {                       {


    x = x + 1;              x = x + 2;              x = x * 2;


}                       }                       }
```

Add P(), V() semaphore operations (using semaphores s1, s2, s3) in the code for thread 1, 2 and 3 such that the concurrent execution of the three threads can only result in the value of x = 6.