**Name:**_______________________________________

**Student ID:**_______________________________________

# CS230 System Programming

# Midterm Exam

Tuesday, October 25, 2011 1:00-3:00pm

**Instructions:**

- Make sure that your exam is not missing any sheets, then write your full name 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 90 points.
- This exam is CLOSED BOOK, CLOSED NOTES. You may only have two sheets of paper or notes with you. Good luck!

| | |
|---|---|
| 1 (12): | |
| 2 (13): | |
| 3 (12): | |
| 4 (10): | |
| 5 (8): | |
| 6 (15): | |
| 7 (10): | |
| 8 (10): | |
| TOTAL (90): | |

## Problem 1. (12 points):

Consider the following $8$-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is $2^{3-1} - 1 = 3$.
- The last 4 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where $M$ is the significand and $E$ is the biased exponent.

The rules are like those in the IEEE standard(normalized, denormalized, representation of $0$, infinity, and NAN). FILL in the table below. Here are the instructions for each field:

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form $x$ or $\frac{x}{y}$, where $x$ is an integer, and $y$ is an integral power of $2$. Examples include $0$, $\frac{3}{4}$.
- **E:** The integer value of the exponent.
- **Value:**The numeric value represented.

  Note: you need not fill in entries marked with ”—”.

| Description | Binary | $M$ | $E$ | Value |
|---|---|---|---|---|
| Minus zero | | | | $-0.0$ |
| — | `0 100 0101` | | | |
| Smallest denormalized (negative) | | | | |
| Largest normalized (positive) | | | | |
| One | | | | 1.0 |
| — | | | | 5.5 |
| Positive infinity | | — | — | $+\infty$ |

## Problem 2. (13 points):

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
  return mat1[i][j] +  mat2[i][j];
}
```

A. Suppose the above code generates the following assembly code:

```
sum_element:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%ecx
        sall $2,%ecx
        leal 0(,%eax,8),%edx
        subl %eax,%edx
        leal (%eax,%eax,4),%eax
        movl mat2(%ecx,%eax,4),%eax
        addl mat1(%ecx,%edx,4),%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

What are the values of `M` and `N`?

`M` =

`N` =

## Problem 3. (12 points):

Condider the following assembly code for a C `for` loop:

```
loop:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%ecx
        movl 12(%ebp),%edx
        xorl %eax,%eax
        cmpl %edx,%ecx
        jle .L4
.L6:
        decl %ecx
        incl %edx
        incl %eax
        cmpl %edx,%ecx
        jg .L6
.L4:
        incl %eax
        movl %ebp,%esp
        popl %ebp
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables `x`, `y`, and `result` in your expressions below — *do not use register names.*)

```
int loop(int x, int y)
{
   int result;

   for (_____________; ___________; result++ ) {

      __________;

      __________;
   }

   __________;

   return result;
}
```

## Problem 4. (10 points):

Match each of the assembler routines on the left with the equivalent C function on the right.

```
foo1:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     sall $4,%eax
     subl 8(%ebp),%eax
     movl %ebp,%esp
     popl %ebp
     ret

foo2:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     testl %eax,%eax
     jge .L4
     addl $15,%eax
.L4:
     sarl $4,%eax
     movl %ebp,%esp
     popl %ebp
     ret

foo3:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     shrl $31,%eax
     movl %ebp,%esp
     popl %ebp
     ret
```

```
int choice1(int x)
{
    return (x < 0);
}

int choice2(int x)
{
    return (x << 31) & 1;
}

int choice3(int x)
{
    return 15 * x;
}

int choice4(int x)
{
    return (x + 15) /4
}

int choice5(int x)
{
    return x / 16;
}

int choice6(int x)
{
    return (x >> 31);
}
```

**Fill in your answers here:**

foo1 corresponds to choice ________.

foo2 corresponds to choice ________.

foo3 corresponds to choice ________.

The next problem concerns the following C code:

```
/* copy string x to buf */
void foo(char *x) {
  int buf[1];
  strcpy((char *)buf, x);
}

void callfoo() {
  foo("abcdefghi");
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
080484f4 <foo>:
080484f4: 55              pushl  %ebp
080484f5: 89 e5           movl   %esp,%ebp
080484f7: 83 ec 18        subl   $0x18,%esp
080484fa: 8b 45 08        movl   0x8(%ebp),%eax
080484fd: 83 c4 f8        addl   $0xfffffff8,%esp
08048500: 50              pushl  %eax
08048501: 8d 45 fc        leal   0xfffffffc(%ebp),%eax
08048504: 50              pushl  %eax
08048505: e8 ba fe ff ff  call   80483c4 <strcpy>
0804850a: 89 ec           movl   %ebp,%esp
0804850c: 5d              popl   %ebp
0804850d: c3              ret

08048510 <callfoo>:
08048510: 55              pushl  %ebp
08048511: 89 e5           movl   %esp,%ebp
08048513: 83 ec 08        subl   $0x8,%esp
08048516: 83 c4 f4        addl   $0xfffffff4,%esp
08048519: 68 9c 85 04 08  pushl  $0x804859c   # push string address
0804851e: e8 d1 ff ff ff  call   80484f4 <foo>
08048523: 89 ec           movl   %ebp,%esp
08048525: 5d              popl   %ebp
08048526: c3              ret
```

## Problem 5. (8 points):

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '`\0`' character) to address `dst`. It does **not** check the size of the destination buffer.
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

| Character | Hex value | Character | Hex value |
|---|---|---|---|
| 'a' | 0x61 | 'f' | 0x66 |
| 'b' | 0x62 | 'g' | 0x67 |
| 'c' | 0x63 | 'h' | 0x68 |
| 'd' | 0x64 | 'i' | 0x69 |
| 'e' | 0x65 | '\0' | 0x00 |

Now consider what happens on a Linux/x86 machine when `callfoo` calls `foo` with the input string "`abcdefghi`".

A. List the contents of the following memory locations immediately after `strcpy` returns to `foo`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

`buf[0] = 0x`____________________

`buf[1] = 0x`____________________

`buf[2] = 0x`____________________

B. Immediately **before** the `ret` instruction at address `0x0804850d` executes, what is the value of the frame pointer register `%ebp`?

`%ebp  = 0x`____________________

C. Immediately **after** the `ret` instruction at address `0x0804850d` executes, what is the value of the program counter register `%eip`?

`%eip  = 0x`____________________

## Problem 6. (15 points):

Consider the following incomplete definition of a C struct along with the incomplete code for a function `func` given below.

```
typedef struct node {

  ________________ x;

  ________________ y;

  struct node *next;

  struct node *prev;

} node_t;
```

```
node_t n;

void func() {

  node_t *m;

  m = ________________________;

  m->y /= 16;

  return;
}
```

When this C code was compiled on an IA-32 machine running Linux, the following assembly code was generated for function `func`.

```
func:
  pushl %ebp
  movl n+12,%eax
  movl 16(%eax),%eax
  movl %esp,%ebp
  movl %ebp,%esp
  shrw $4,8(%eax)
  popl %ebp
  ret
```

Given these code fragments, fill in the blanks in the C code given above. Note that there is a unique answer.

The types must be chosen from the following table, assuming the sizes and alignment given.

| Type | Size (bytes) | Alignment (bytes) |
|---|---|---|
| char | 1 | 1 |
| short | 2 | 2 |
| unsigned short | 2 | 2 |
| int | 4 | 4 |
| unsigned int | 4 | 4 |
| double | 8 | 4 |

## Problem 7. (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 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

| 2-way Set Associative Cache | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | Tag | Valid | Byte 0 | Byte 1 | Byte 2 | Byte 3 | Tag | Valid | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
| 0 | 09 | 1 | 86 | 30 | 3F | 10 | 00 | 0 | 99 | 04 | 03 | 48 |
| 1 | 45 | 1 | 60 | 4F | E0 | 23 | 38 | 1 | 00 | BC | 0B | 37 |
| 2 | EB | 0 | 2F | 81 | FD | 09 | 0B | 0 | 8F | E2 | 05 | BD |
| 3 | 06 | 0 | 3D | 94 | 9B | F7 | 32 | 1 | 12 | 08 | 7B | AD |
| 4 | C7 | 1 | 06 | 78 | 07 | C5 | 05 | 1 | 40 | 67 | C2 | 3B |
| 5 | 71 | 1 | 0B | DE | 18 | 4B | 6E | 0 | B0 | 39 | D3 | F7 |
| 6 | 91 | 1 | A0 | B7 | 26 | 2D | F0 | 0 | 0C | 71 | 40 | 10 |
| 7 | 46 | 0 | B1 | 0A | 32 | 0F | DE | 1 | 12 | C0 | 88 | 37 |

## 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**: `0E34`

A. Physical address format (one bit per box)

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

B. Physical memory reference

| Parameter | Value |
|---|---|
| Byte offset | 0x |
| Cache Index | 0x |
| Cache Tag | 0x |
| Cache Hit? (Y/N) | |
| Cache Byte returned | 0x |

## Problem 8. (10 points):

After watching the mayor election, you decide to start a business in developing software for electronic voting. The software will run on a machine with a 1024-byte direct-mapped data cache with 64 byte blocks (i.e., cache line size is 64 bytes).

You are implementing a prototype of your software that assumes that there are 7 candidates. The C-structures you are using are:

```
struct vote {
    int candidates[7];
    int valid;
};

struct vote vote_array[16][16];
register int i, j, k;
```

You have to decide between two alternative implementations of the routine that initializes the array `vote_array`. You want to choose the one with the better cache performance.

You can assume:

- `sizeof(int) = 4`
- `vote_array` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `vote_array`. Variables `i`, `j` and `k` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        vote_array[i][j].valid=0;
    }
}

for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        for (k=0; k<7; k++) {
            vote_array[i][j].candidates[k] = 0;
        }
    }
}
```

Total number of misses in the first loop: _________ %

Total number of misses in the second loop: _________ %

Overall miss rate for writes to `vote_array`: _________ %

B. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        for (k=0; k<7; k++) {
            vote_array[i][j].candidates[k] = 0;
        }
        vote_array[i][j].valid=0;
    }
}
```

Miss rate for writes to `vote_array`: _________ %